Performance of Multithreaded Chip Multiprocessors And Implications for Operating System Design

Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design,

Alexandra Fedorova,
Margo Seltzer,
Christopher Small
and
Daniel Nussbaum

TR-09-05

Computer Science Group
Harvard University
Cambridge, Massachusetts

Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design

Alexandra Fedorova[†‡], Margo Seltzer[†], Christopher Small[‡] and Daniel Nussbaum[‡]
[†]*Harvard University,* [‡]*Sun Microsystems*

Abstract

An operating system's design is often influenced by the architecture of the target hardware. While uni-processor and multiprocessor architectures are well understood, such is not the case for multithreaded chip multiprocessors (CMT) – a new generation of processors designed to improve performance of memory-intensive applications. The first systems equipped with CMT processors are just becoming available, so it is critical that we now understand how to obtain the best performance from such systems.

The goal of our work is to understand the fundamentals of CMT performance and identify the implications for operating system design. We have analyzed how the performance of a CMT processor is affected by contention for the processor pipeline, the L1 data cache, and the L2 cache, and have investigated operating system approaches to the management of these performance-critical resources. Having found that contention for the L2 cache can have the greatest negative impact on processor performance, we have quantified the potential performance improvement that can be achieved from L2-aware OS scheduling. We evaluated a scheduling policy based on the balance-set principle and found that it has a potential to reduce miss ratios in the L2 by 19-37% and improve processor throughput by 27-45%. To achieve a similar improvement in hardware requires doubling the size of the L2 cache.

1. INTRODUCTION

An operating system provides a layer of abstraction between the hardware and the software. Its job is to expose the power of the hardware to applications, while hiding its complexities. It is no surprise that the architecture of the hardware influences the design of the operating system. The subject of operating system design for conventional processors has been addressed in the past. This paper begins to investigate the area of operating system design for a new family of processors: multithreaded chip multiprocessors (CMT).

CMT processors combine chip multiprocessing (CMP) and hardware multithreading (MT) – today's architectural trends designed to improve processor utilization by offering better support for thread-level parallelism. 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 interleaves execution of instructions from different threads. As a result, if one thread blocks on a memory access, other threads can make forward progress. Numerous studies have demonstrated the performance benefits of CMP and MT [4-7, 13, 23, 29, 36].

Thanks to improvement in chip densities, it has become possible to combine these two approaches in CMTs. The goal of CMTs is to improve performance of an important class of modern applications, such as web services, application servers, and on-line transaction processing systems. These applications are notorious for poor utilization of the CPU pipeline – they usually include multiple threads of control executing short sequences of integer operations, with frequent dynamic branches. Such 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% [14]. This means that the majority of the time, the processor pipeline is being unused. CMT processors are designed to address this problem.

Systems equipped with CMT processors are beginning to become available at the time of the writing. IBM released its POWER 5 CMT chip in the summer of 2004 [18]; Sun Microsystems and Intel plan to ship their commercial CMT processors in 2005 [8, 9].

CMTs differ from conventional processors in a fundamental way: they are equipped with dozens of simultaneously active thread contexts (e.g., Sun's Niagara processor will have eight cores, each with four thread contexts [8]), and, as a result the competition for shared resources is intense. This carries new implications for operating system designs targeted at

such processors. We have studied how contention for the processor pipeline, the L1 data cache and the L2 cache affects system performance, with the objective of understanding which of these shared resources are most likely to become performance bottlenecks so that we can adapt operating systems to compensate for these bottlenecks. We have found that the latency resulting from a poor hit rate in the L1 cache can be effectively hidden by hardware multithreading, but that high contention for the L2 can significantly hurt the overall processor performance. This result drove us to investigate how much potential there is in using the OS scheduling to improve L2 (and subsequently overall processor) performance.

We have considered an OS scheduling algorithm based on the balance-set principle [21] and found that it has the potential to reduce the L2 cache miss ratios by 19-37%, yielding a performance improvement of 27-45% – an improvement that could have been achieved in hardware only by doubling the size of the L2 cache.

While this result is encouraging, it is not yet clear what fraction of this potential can be realized by an implementation; at the conclusion of this paper we present a research agenda for implementing this algorithm and discuss the challenges involved in performing this task.

The major contributions of our work include: quantifying the effects of contention for various components of CMT processors on the overall performance, the design of a new scheduling algorithm that yields better L2 performance, an evaluation of this algorithm, and an adaptation of an existing model for cache miss ratios to make it work with multithreaded workloads (this was necessary for evaluation of the balance-set scheduling algorithm).

Using software mechanisms to best exploit existing hardware 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 present-day workloads, accurately predicting the future is a black art. Since hardware cannot adapt as quickly as applications change, it is important to design operating system software that will keep systems running well as 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 Section 3, we analyze shared resources on a CMT processor to identify the performance bottlenecks and conclude that performance optimizations targeted at the L2 cache are likely to have the greatest impact. In Section 4 we investigate potential performance gains from balance-set scheduling for improving the L2 performance. We present the challenges involved in implementing such a scheduler and lay out a future research agenda in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND AND SIMULATOR

The systems addressed in this study have multiple processor cores on a chip (chip multiprocessing [36]) and multiple hardware thread 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 processor switches among thread contexts.

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 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 several hardware thread contexts (usually 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. When one or more threads are blocked, the system continues to switch among the remaining available threads.

For the purposes of our study we built a CMT system simulator toolkit [16] as a set of extensions to the Simics simulation toolkit [15]. Simics provides full-system simulation of several popular hardware platforms. Our simulator is based on an UltraSPARC II® machine. Simics can bootstrap the simulated machine with the Solaris™ operating system and standard Unix environment. All the simulations described in this paper are execution-driven and include both user-level and OS code.

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. We have decided to simulate a simple, classical RISC core, as opposed to a complex out-of-order processor, because we believe that this is a viable architecture for future CMT processors with a large number of processor cores on a chip. Allowing each core to be simple, allows placing more multithreaded cores on a chip – and this is an attractive design option for OLTP and server workloads, since such workloads typically have a high degree of application-level parallelism and benefit little from complex super-scalar pipeline designs. Moreover, a previous study has shown that for such workloads, core complexity should be traded off for an increased number of hardware contexts [24]. Additionally, 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.

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 configured with one to four processor cores, depending on the experiment. Each core includes four hardware contexts, an 8KB L1 data cache and a 16KB L1 instruction cache (both 4-way set-associative). We simulate a unified 12-way set-associative L2 cache, shared among all cores on a chip, 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. SOURCES OF PERFORMANCE BOTTLENECKS

In this section we analyze processor pipeline and on-chip caches, L1 data cache and the L2 cache, as potential performance bottlenecks on CMT processors. Based on our experience, other shared resources such as the TLB or the L1 instruction cache are not likely to become performance bottlenecks for the workloads we consider, and, therefore, we do not address them in this study.

Multithreaded processors are specifically designed to hide memory latency experienced by applications when the processor cache hit rate is poor. When memory latency can be hidden by hardware multithreading, the processor pipeline becomes a performance bottleneck. However, when memory latency is too high to be hidden, the on-chip caches become the performance bottleneck.

This section consists of two parts: first, in Section 3.1, we consider the scenario when the processor pipeline is the performance bottleneck. We propose an OS scheduling technique designed to better manage pipeline contention. In Section 3.2 we investigate the scenario when the processor caches are the performance bottleneck. Specifically, we explore under which conditions the L1 data cache (Section 3.2.1) or the L2 cache (Section 3.2.2) become performance bottlenecks. We conclude that while hardware multithreading does an excellent job of hiding latency resulting from faults in the L1 data cache, its capability to hide latencies resulting from poor performance in the L2 is much more limited. In Section 4 we propose an OS scheduling algorithm designed to manage L2 contention.

3.1. Processor pipeline

Operating system approaches to managing pipeline contention in multithreaded processors have been proposed before [1, 2, 12]. As we elaborate in Section 6, we were concerned that the applicability of these approaches is limited for CMT processors because they will not scale well in systems with several dozens of hardware contexts. We investigated an approach with better scalability properties.

They key observation is that 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. Threads that exhibit poor locality of memory reference, such as database applications, frequently stall while waiting for a response from the memory hierarchy: such threads are *memory-intensive*.

This observation suggests how to better manage pipeline contention. By co-scheduling compute-intensive threads with memory-intensive threads on the same processor core the scheduler can balance the demand for pipeline resources across cores. A compute-intensive thread can use the pipeline while the memory-intensive thread is blocked on memory. 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 identify compute-intensive and memory-intensive threads by measuring the workload's *CPI* (cycles per instruction) metric. Workloads with low CPIs (those nearing 1 on our simple pipeline) have high pipeline utilization, and vice versa. Using CPI as a

heuristic for scheduling is especially attractive on a CMT processor: CPI can be easily measured, nicely captures the workloads' pipeline utilization, and enables scheduling decisions based on local information.

We have qualified the potential performance improvement from such a scheduling policy, by running the following experiment. 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. To assign threads to a particular processor, we use a `processor_bind()` system call, available on Solaris™. Each thread assigned to a particular core is bound to its own hardware context. The processor interleaves instructions issued on processors' contexts. Therefore, all threads run in parallel and get equal shares of processor issue slots. We start the measurement at the main loop of the benchmarks and stop the measurement whenever any thread finishes.

Table 1 illustrates four different schedules that we evaluated. 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.

	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 1 shows the IPC (instructions per cycle) achieved by each schedule, broken down by CPU. As expected, schedules a) and (b) achieve the best performance[1]. The difference in IPC between schedule (d) and schedules (a) and (b) is a factor of two. The IPC breakdown by CPU indicates that the reason for better performance for schedules (a) and (b) is that they produce a more balanced utilization of processor resources.

While the potential performance improvement from this scheduling technique is dramatic, to achieve it, it was necessary to have threads with a wide range of CPIs. To determine whether real workloads can benefit from this scheduling technique in the same way we measured single-threaded CPIs for a number of standard integer benchmarks: SPEC CPU (int), SPEC JVM, SPEC JBB and SPEC Web. We saw a very small variation in CPIs across the benchmarks. For most benchmarks, the CPI hovers around 4. The smallest average CPI that we saw was 2.37 for 164.gzip, the largest – 5.11 for 181.mcf. The dynamic instruction mix also shows little variation across the benchmarks. If these benchmarks are characteristic of real workloads, these data show that there is little variation in pipeline utilization. Therefore, there is little potential for performance gains from exploiting this variation. Our experiments with the SPEC benchmarks confirmed that performance gains from CPI-based scheduling are modest (about 5%) for such workloads.

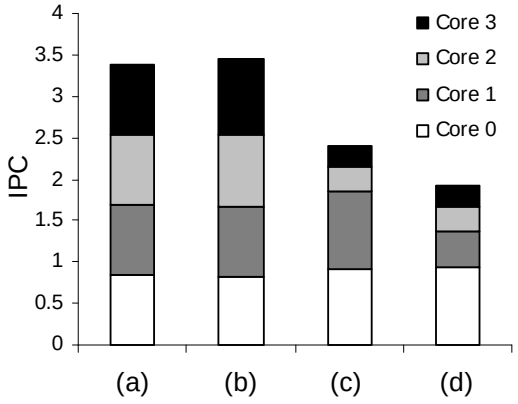

Figure 1. IPC achieved by each schedule, broken down by CPU.

While the benefits of CPI-based scheduling for SPEC workloads are small with our simulated 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 may be greater on such machines. Since our simulator does not have the ability to simulate a super-scalar pipeline, we could not confirm that this is the case. But this is an interesting area for future exploration.

3.2. Processor caches

3.2.1. The 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 8KB [13]. We were concerned that this would result in high latencies

[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.

associated with handling cache faults, which the multi-threaded processor would not be able to hide. The latency-hiding ability depends on the degree of multithreading in the processor's core. We use four for the degree of multithreading, which has been shown to work best for the architecture we are simulating [6]. For the experiments in this section, the simulator has been configured with a single core: since there is an L1 data cache on each core, simulating a single core was sufficient.

We measured cache miss ratios for several cache sizes for the following workloads: SPEC CPU's 186.crafty, 164.gzip, 175.vpr (place), Berkeley DB and SPEC JBB. For each experiment, we ran four copies of the same benchmark on a single-core machine – the threads running the workload shared the core's data cache. Benchmarks did not share any data. We experimented with several cache sizes, from 8KB (this would be a typical cache size used on a CMT core) to 128 KB.

As Figure 2 shows, for small cache sizes, cache miss ratios are way above what is considered acceptable for conventional single-threaded processors. For SPEC JBB, the miss ratio is high even when the cache size is 128 KB.

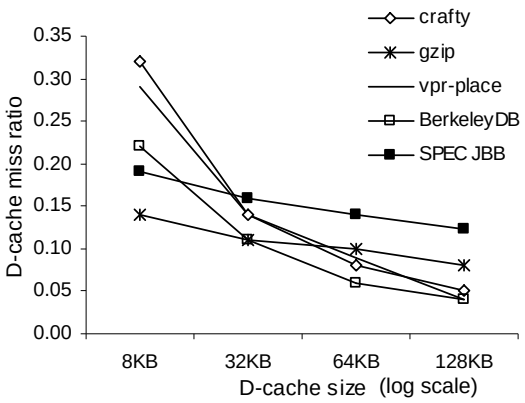

Figure 2. D-cache miss ratios as the cache size changes.

However, as we can see from Figure 3, the processor performance is immune to such poor cache behavior. Even when cache miss ratios are above 20%, the IPC[2] (instructions per cycle) is between 0.7 and 0.9, and it does not change much as cache size changes. This suggests that hardware multithreading does a good job of hiding the latency associated with faulting in the L1 data cache. The implication of this result is that operating system policies targeted at improving performance in the L1 data cache are not likely to bring significant performance improvements (and we have confirmed this experimentally), because the hardware already masks poor cache performance.

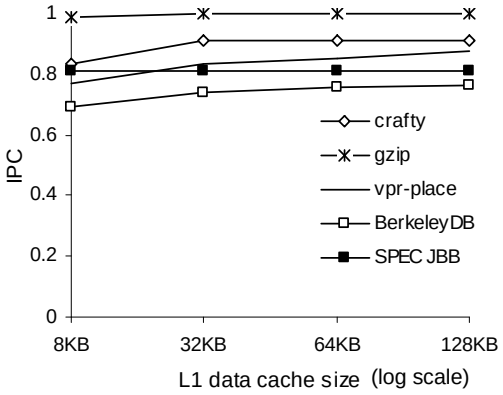

Figure 3. Pipeline utilization as D-cache size changes.

3.2.2. The L2 cache

The L2 cache has a greater potential for becoming a performance bottleneck when the miss ratio is high, because the latency between the L2 and main memory is significantly greater than that between the 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 20 and 120 cycles in our model. Penalties from faulting in the L2 are further increased due to competition for main memory bandwidth.

To evaluate the effect of L2 performance on processor IPC, we used a dual-core MT machine with four thread contexts per core, an 8KB L1 data cache and L2 caches of varying sizes.

Our workload consists of nine benchmarks from the SPEC CPU 2000 suite (164.gzip, 175.vpr (place), 175.vpr (route), 176.gcc, 179.art, 186.crafty, 188.ammp, 197.parser, 255.vortex). We chose these particular benchmarks, because we wanted our workload to include programs with both good and poor cache locality. We run two copies of each benchmark, giving us an 18-thread workload (each benchmark is a single-threaded process that runs in its own thread).

As explained in Section 2, our simulations are execution-driven, and the benchmarks are run in the Solaris™ 9 operating environment. Solaris™ 9 does not include special support for CMT processors. From the point of view of the OS, our simulated dual-core four-way multithreaded machine looks like a conventional eight-way multiprocessor. The Solaris™ scheduler assigns threads to hardware contexts as if it were assigning them to processors on a multiprocessor system, picking eight threads at a time to schedule on

[2] Our simple pipeline issues at most one instruction each cycle. The highest IPC it is able to obtain is one.

the eight hardware contexts during each scheduling time slice. We vary the size of the L2 cache and show how this affects L2 performance and processor IPC. 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. We fast-forward the simulation until all running threads reach the main loop, and then perform a detailed simulation for one billion cycles.

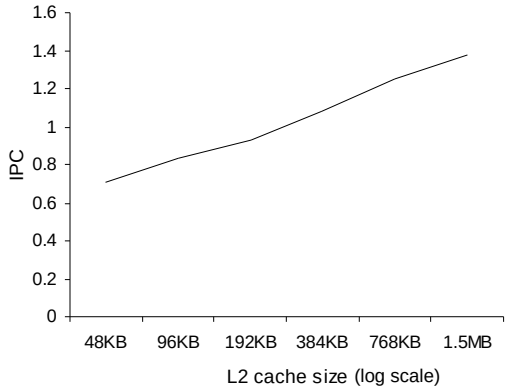

Figure 4. IPC for the 18-process SPEC workload**.** Processor IPC is sensitive to the size of the L2.

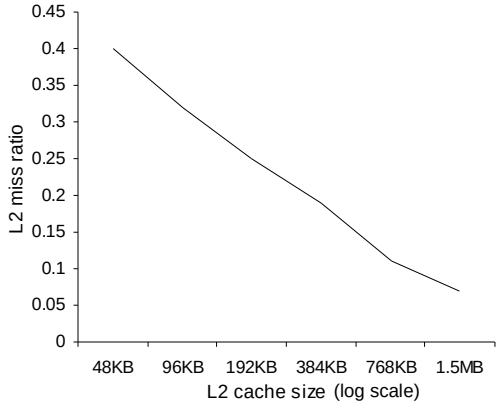

Figure 5. L2 miss ratios for the 18-process SPEC workload. The L2 miss ratio falls as the L2 size increases.

Figure 4 shows the processor IPC resulting from running this SPEC workload. Performance degradation is evident as the L2 cache becomes smaller. Figure 5 shows that the reason for performance degradation is high miss ratios for small L2 sizes.

This result suggests that multithreaded processors are limited in their ability to hide high latency resulting from handling faults in the L2. Since modern applications exhibit a dangerous trend of becoming progressively more data-intensive, it is possible that CMT processors equipped with a sufficiently large L2 cache for today's workloads will fail to satisfy the needs of applications in the near future. Since changing software is easier than changing hardware, it is wise to equip the operating system with the ability to handle resource shortages when the hardware fails to do so. The next section quantifies the potential performance improvement that can be achieved from better OS scheduling.

4. BALANCE-SET SCHEDULING

Balance-set scheduling has been proposed by Denning [20] as a way of improving the performance of virtual memory. We evaluated the effectiveness of this approach for the L2 cache. The idea behind balance-set scheduling is as follows. Separate all runnable threads in subsets, or groups, such that the combined working set of each group fits in the cache. Then, schedule a group at a time for the duration of the scheduling time slice. By making sure that the working set of each scheduled group fits in the cache, this algorithm is designed to reduce cache miss ratios.

However, we found that working set size was not a good indicator of the workload's cache behavior. When we used a model based on the working set principle [22] to estimate cache miss ratios of SPEC workloads, the estimates were often inconsistent: a workload with a large working set would produce lower miss ratios than a workload with a small working set, and vice versa. Further investigation [39] revealed that the working set is a good indicator of a workload's cache miss ratio only if the program accesses its working set uniformly – this assumption of uniform access is the premise underlying Denning's original work on balance-set scheduling [21]. As we learned, this assumption does not hold for standard benchmarks (SPEC CPU, SPEC Web, SPEC JBB).

We needed to find a better way to assess a workload's cache behavior, so we searched for a model that would accurately estimate cache miss ratios based on some characteristics of a workload.

In Section 4.1 we describe the model for cache miss ratio that we used and how we adapted this model so that it could be used with balance-set scheduling. In Section 4.2 we evaluate the potential performance gains from the balance-set scheduling algorithm based on this model.

4.1. A model for cache miss ratios

Berg and Hagersten developed a model for estimating cache miss ratios based on *reuse distances* [19]. A reuse distance is the amount of time[3] that passes

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

between successive references to the same memory location. The model bases the probability of a cache hit on the reuse distance. The smaller the reuse distance, the greater the probability that the reference results in a hit. Using this model, Berg and Hagersten estimated cache miss ratios of SPEC workloads to within about 5% of actual values.

Figure 6 shows an example of a reuse-distance histogram (a non-normalized reuse-distance distribution) for 188.ammp – a benchmark from the SPEC CPU 2000 suite. A significant fraction of the reuse distances are large – this is an indication of poor locality. Our experiments confirmed that this benchmark has exceptionally high cache miss ratios.

The input necessary for this model, a reuse-distance histogram, can be built at runtime by monitoring memory references, using the standard hardware watchpoint mechanism [19].

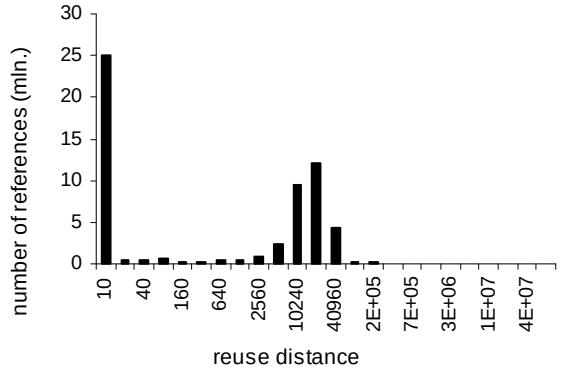

Figure 6. Reuse distance distribution for 188.ammp.

The reuse-distance model estimates cache miss ratios for a single workload. For balance-set scheduling, we need to decide which threads produce the lowest miss ratios when scheduled to run together and access the cache concurrently. Therefore, we need to estimate cache miss ratios for groups of threads. To do this, we build individual reuse-distance histograms for threads, and then combine these histograms to estimate miss ratios for groups of threads.

We developed three methods for estimating group miss ratios: COMB, COMB+IPC, and AVG. COMB combines individual reuse distance histograms into a single histogram and uses this histogram to predict the miss ratio for the group. COMB+IPC improves on COMB by making adjustments for relative differences in the speeds of the running threads. AVG estimates miss ratios for each thread in a group as if it ran with a dedicated partition of a smaller cache and then averages the resulting ratios. We will now describe these methods in detail and then evaluate their accuracy.

4.1.1. Method #1: COMB

To combine several histograms into one, we use the following two-step algorithm:

1. For each reuse distance, sum the number of references falling within this distance 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). See Figure 7.

Step 2 is necessary because when several threads share the cache using fine-grained multithreading, 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 could intervene with their own memory references. As a result, the original reuse distance of one, for instance, could become anywhere between one and four. We have experimented with adjusting the reuse distance by multiplying it 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 worked better for larger caches, and a coefficient of N worked better for smaller caches.

Histogram A	
R.dist	**# ref.**
1	90
10	50
20	78
…	…
100	14

Histogram B	
R.dist	**# ref.**
1	30
10	46
20	27
…	…
100	18

Histogram A+B	
R.dist	**# ref.**
2	120
20	96
40	105
…	…
200	32

Figure 7. Combining reuse-distance histograms.

Our next method, COMB+IPC, deterministically calculates this coefficient, by taking into account the different rates at which threads issue memory references.

4.1.2. Method #2: COMB + IPC

When several threads share a cache, the cache access patterns of slower threads will have a smaller effect on the overall cache miss ratio than those of fast threads, because slower threads will issue memory references at a slower rate. It is necessary to account for these differences when combining reuse-distance histograms.

The memory reference rate, number of references per cycle, can be decomposed into two components,

references per instruction and instructions per cycle (IPC):

*Refs. per cycle = Refs per instr. * IPC.*

To account for relative differences in references per instruction, we collect the data for reuse-distance histograms over the same window of instructions for all threads.

To understand how we adjust for relative differences in instructions per cycle (IPC), consider the following example:

Suppose FastThread runs twice as fast as SlowThread – their respective IPCs are 1 and 0.5. When these two threads run together, about half of FastThread's references will be interspersed with SlowThread's references. Therefore, for FastThread, half of the reuse distances will remain the same, and the other half will increase by a factor of two. (See Figure 8.) To reflect this, we multiply the reuse distances in FastThread's histogram by the following coefficient: (0.5 * 1 + 0.5 * 2) = 1.5

SlowThread, however, will not be so lucky: all of its memory references will be interspersed with the references made by FastThread. In fact, each reference that SlowThread makes will be interspersed with two memory references made by FastThread. Therefore, all reuse distances will increase by a factor of three.

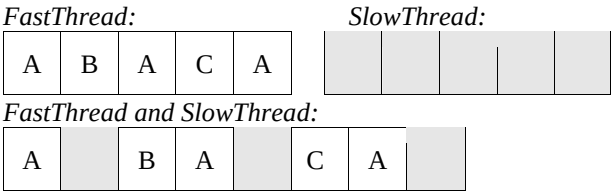

Figure 8. Illustration of how reuse distances change when threads with unequal speeds run in parallel.

These observations lead us to the following formula for DIST_COEFF, a coefficient by which we adjust the values of reuse distances of each thread:

$$DIST\_COEFF_i = \left(\sum_{j=1}^{n} IPC_j \right) / IPC_i ,$$

where *i* is the index of the thread whose coefficient we are computing and *n* is the total number of threads in the group.

Finally, we need to make an adjustment to the number of references that fall within each reuse-distance: a slow thread will be issuing memory references more slowly than a fast thread, so we need to scale the number of references that fall within each reuse-distance accordingly. This produces the following formula to compute REFS_COEFF – the corresponding scaling coefficient:

$$REFS\_COEFF_i = IPC_i / MAX\_IPC,$$

where *i* is the index of the thread whose coefficient we compute and MAX_IPC is the largest IPC in the combination of threads.

Once we adjust each reuse distance histogram by multiplying reuse distances by *DIST_COEFF* and the number of references by *REFS_COEFF*, we simply merge all reuse distance histograms (See Figure 9).

Histogram A (IPC = 1)	
R.dist	**# ref.**
1.5	90
15	50
30	78
…	…
150	14

Histogram B (IPC = 0.5)	
R.dist	**# ref.**
3	15
30	23
60	14
…	…
300	9

Histogram A+B	
R.dist	**# ref.**
1.5	90
3	15
15	50
…	…
300	9

Figure 9. Combining adjusted reuse-distance histograms. Note that we use the same histograms as in the example in Figure 7, and make the adjustments to them according to the method described in this section.

The drawback of a combination-based approach such as this one is that it is too computationally expensive to implement in a real system. Imagine a machine with 32 thread contexts and 100 threads. The scheduler has to potentially combine $\binom{100}{32}$ histograms, and estimate the miss ratios. Although this computation does not have to be performed often (as we explain in Section 6), it still adds to the overhead. The approach we describe next is more practical.

4.1.3. Method #3: AVG

The intuition behind AVG is to pretend that each thread runs with its own dedicated partition of a cache. AVG estimates miss ratios for each thread as if it ran with a smaller cache, of size TOTAL_CACHE / NUMBER_OF_THREADS. Then, to predict the miss ratio for a group of threads, it averages the estimated ratios of the individual threads. We found that predictions based on this method are just as accurate as those based on combination-based methods, and are significantly less expensive to produce.

The miss ratio estimate has to be computed once for each workload, and then, to project miss ratios for multiple workloads, the scheduler needs only to average

several values. Furthermore, the scheduler can remember averages for small groups of threads, and then predict the miss ratio for a larger group by merging the known averages.

4.1.4. Evaluating Model Accuracy

To test the effectiveness of the model, we used the 18-benchmark SPEC CPU2000 workload described in Section 3.2.2. We estimated miss ratios for all possible groups consisting of eight threads, for four L2 cache sizes, using COMB, COMB+IPC and AVG. We validated the estimated miss ratios for those thread groups that produced the best schedule for the balance-set scheduling algorithm, for each cache size. (We explain how such groups were selected in the next section.)

To validate the estimated miss ratios for the groups, we simulated each group on a machine with two cores and four hardware contexts per core for 100 million cycles after fast-forwarding the simulation past the benchmark initialization phase.

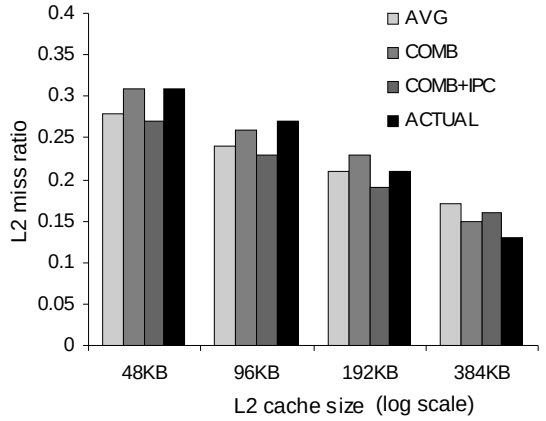

Figure 10. Actual vs. predicted miss ratios.

Figure 10 shows the actual and predicted miss ratios obtained using the three methods for various cache sizes. The displayed miss ratios are the averaged quantities for the thread groups producing the best schedule for each cache size (standard deviation is small). Our estimated miss ratios are, on average, within 17% of the actual miss ratios. Errors are expected, given that we applied the model to a multi-program workload, simulated a set-associative cache (the reuse-distance model assumes a fully-associative cache), and used a window size of 500 million instructions when collecting data for reuse-distance histograms (Berg and Hagesten suggest using smaller windows in order to avoid spanning different phases in behavior of a program).

Although error rates are high, reducing them is not critical for us: what is important is that the model is accurate enough to distinguish between thread groups that produce high miss ratios and those that produce low miss ratios. And, as we demonstrate in the next section, this is precisely what we need to improve the performance of the default scheduler.

4.2. Balance-set scheduling

The ability to predict cache miss ratios for groups of threads makes it possible to identify the thread groups that produce low cache miss ratios. We leverage this in the balance-set scheduling algorithm. Before implementing the algorithm, we wanted to quantify the performance improvement that could be expected from using it. This section describes how we evaluated the potential of this algorithm for reducing L2 contention and improving processor performance by emulating its actions. First, we describe the scheduling algorithm, and then explain how we emulated it. In Section 4.3 we present performance results and offer analysis. We discuss implementation challenges and potential runtime costs in Section 5.

4.2.1. Scheduling Algorithm

Step 1 – *Choosing the L2 miss ratio threshold* (Performed periodically, as the set of threads running on the system changes):

> The job of the scheduler will be to keep the L2 miss ratios below this threshold. The goal is to set the miss ratio threshold to be as low as possible, but not too low so as to starve some threads. The scheduler first identifies the most cache-greedy thread, because it is this thread that runs the danger of being starved. Miss ratios obtained from analysis of individual reuse-distance histograms are used to identify this thread.
>
> Next, once the scheduler has analyzed the miss ratios for all groups of threads, it examines the estimated miss ratios for the groups that include the greediest thread, and picks the smallest miss ratio. This miss ratio is the smallest that can be achieved without starving the greediest thread, and it is this miss ratio that is chosen as the threshold.

Step 2 – *Miss ratio analysis to identify the thread groups that will produce low cache miss ratios* (Performed periodically, as the set of threads running on the system changes):

> From the entire pool of runnable threads, construct thread groups, such that the number of threads in each group is equal to the number of the available hardware contexts. Estimate L2 miss ratios for all groups, using the reuse-distance model and the histogram-combining method AVG. Discard those groups whose estimated miss ratio is above the

threshold. The remaining groups are *candidate groups.*

Step 3 – *Scheduling decision* (performed every time a scheduling time slice expires):

Choose a group from the set of candidate groups. Schedule the threads in the group to run during the current time slice. (Note that a thread may belong to more than one group.) Keep track of how much processor time each thread has received. On the next time slice, make a selection from candidate groups that contain threads that previously received little or no processor time.

4.2.2. Emulating the scheduler

In this section we first explain how we emulated the process of making a scheduling decision and then how we simulated the resulting schedule. We use the 18-thread SPEC workload described in Section 3.2.2 and the CMT processor with two cores and four hardware contexts on each core. We instrumented our simulator to collect the data for reuse-distance histograms and built reuse-distance histograms for all the threads off-line.

4.2.2.1. Making the scheduling decision

First, we examine all $\binom{18}{8}$ combinations of threads – we have a total of 18 threads, but only eight can be scheduled at a time on the processor's eight hardware contexts. We compute group miss ratios for all groups, and sort the groups in ascending order by miss ratio. Next, we pick the miss ratio threshold as described in Step 1 of the scheduling algorithm (Section 4.2.1). The groups whose estimated miss ratio is below the threshold form the *candidate set*.

From the candidate set we pick several groups that will form the final *schedule.* Because each group has eight threads, and the total workload has 18 threads, a schedule needs to include at least three groups, to make sure that each of the 18 threads gets to run. The idea is that each group in a schedule will run for a scheduling time slice; once all groups in the schedule have run, the schedule repeats.

To pick thread groups for a schedule we experimented with two policies: performance-oriented (PERF), and fairness-oriented (FAIR).

With PERF, we select from the candidate set the group with the lowest miss ratio and containing threads have not yet been selected. We repeat this process until each thread is represented in the schedule.

With FAIR, we attempt to equalize the CPU share of each workload: as we select groups of threads we keep track of how many times each thread had been selected. Each time we make a selection, we pick the group that contains the greatest number of the least frequently selected threads.

We evaluate how these policies compare in terms of performance and fairness in Section 4.3.

We repeated the above schedule-construction process for four L2 cache sizes from 48KB to 384KB.

4.2.2.2. Simulation

Once we construct schedules for all cache sizes and policies, we simulate each schedule to measure its performance. For a given schedule, we simulate each of its groups for a time slice (100 million cycles) after fast-forwarding the simulation past the benchmark initialization phase. To obtain the IPC for the schedule we average the IPCs achieved by the groups in that schedule (the standard deviation is small, typically within 5% of the mean). In the same way we compute the L2 miss ratio for the schedule.

Recall that each group contains eight threads – one for each hardware context on the machine. By running a workload of eight CPU-intensive threads on a machine with eight hardware contexts and with nothing else running on that machine, we ensured that each thread ran on its own hardware context. In this way, we emulated the action of a real scheduler that would specifically assign each thread to its own hardware context for a time slice.

Because each thread was assigned to its proper context, and the machine issued instructions from each context in a round-robin fashion, we assured that all threads simulated as a group were given equal shares of processor issue slots and made forward progress.

To measure IPC and L2 miss ratio for the default scheduler, we ran the 18-benchmark workload on the simulated machine, for one billion. cycles, after fast-forwarding the benchmarks past the initialization phase. Threads were dynamically assigned to hardware contexts by the Solaris™ scheduler (recall the explanation in Section 3.2.2).

.

4.3. Results

Figure 11 presents the IPC for each schedule. In all cases, the balance-set scheduler outperforms the default scheduler. The performance gain from the scheduler using the PERF policy is from 27% (384KB L2) to 45% (48KB L2). With the FAIR policy, the performance gain is from 7% (384KB L2) to 34% (48KB L2). As cache pressure becomes greater, there is more benefit to be reaped from the balance-set scheduling approach. As cache contention decreases, there is less performance benefit: if the performance achieved with the default scheduler is already good, there is less room for improvement.

Figure 12 shows the corresponding L2 miss ratios. With balance-set scheduling we were able to reduce the L2 miss rates by 19-37% when using the PERF policy and by 9-18% when using the FAIR policy. 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 48KB L2 cache, the balance-set PERF scheduler achieves the L2 miss ratio of 12%. To achieve a similar miss ratio with the default scheduler, the L2 cache size would have to be 96KB – twice as large. This ratio holds for the remaining cache sizes.

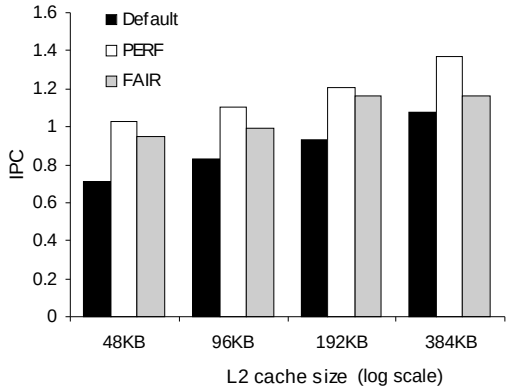

Figure 11. IPC achieved with the default scheduler, and the balance-set scheduler using PERF and FAIR policies.

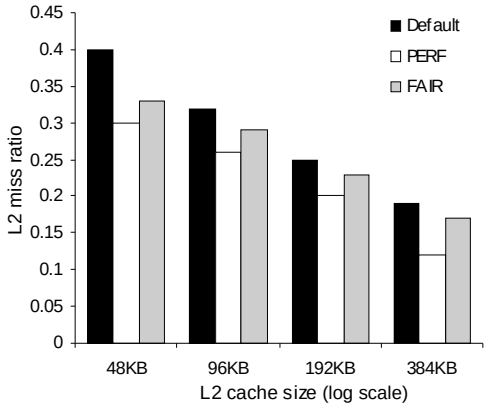

Figure 12. L2 cache miss ratios achieved with the default scheduler, and the balance-set scheduler using PERF and FAIR policies.

The performance improvement that we got resulted from the fact that we have constructed thread groups so that they would share the cache amiably, producing a low L2 miss ratio, and achieving high IPC as a result. Our miss ratio analysis, which preceded schedule construction, helped us avoid scheduling groups of threads that would induce thrashing and waste processor cycles on endless trips to memory.

While this shows that balance-set scheduling has potential to decrease cache contention and improve performance, it is important to address the following questions: Are these results achievable for any workload? Does fairness have to be sacrificed? We provide discussion of these questions in the following section.

4.3.1 Discussion

4.3.1.1. The workload

We used SPEC CPU benchmarks for our experiments because it is a standard workload used for evaluation of CPU performance. We believe that this workload is appropriate for the experiments that stress the memory hierarchy, because this suite of benchmarks has been improved from previous versions specifically for this purpose [3]. It has been modified to include programs whose memory footprints are much larger than traditional cache sizes.

Our workload included benchmarks with good cache locality (164.gzip, 197.parser) as well as poor cache locality (188.ammp, 179.art), so the resulting thread groups had varying cache performance. If, on the contrary, all threads in the workload are identical, all thread groups will produce identical cache performance. In this case, the scheduler may need to schedule fewer threads than available hardware contexts, in order to alleviate the pressure on the L2. This, however, is a double-edged sword: while running fewer threads improves performance in the L2, leaving hardware contexts unused may hurt performance.

From our experience, the payoff from trading better performance in the L2 for unused hardware contexts depends on the workload. In order for the scheduler to decide when it pays to do so, it needs to be able to predict how the L2 performance affects processor IPC. We have developed a model that does this [34] and we plan to incorporate it into the future scheduler implementation.

4.3.1.2. Fairness

Trading off fairness for performance is a canonical issue faced when designing scheduling algorithms [35, 38]. In order to achieve low L2 miss ratio, the job of the balance-set scheduler is to select those groups of threads whose estimated miss ratio is the lowest. Such groups are likely to include cache-frugal threads more often than cache-greedy threads. As a result, cache-frugal threads may get a higher share of CPU. For example, in our experiments a cache-frugal benchmark 197.parser was included in almost every simulation

group, while a cache-greedy 188.ammp was present in only one or two groups per schedule. Whether or not fairness has to be sacrificed depends on the workload.

Although addressing fairness of the balance-set scheduling was not the objective of this study, we attempted to improve fairness with the thread-selection policy FAIR.

To evaluate how FAIR compares to PERF in terms of fairness, we estimated the degree of fairness achieved by these policies using the following metric: standard deviation from the average CPU share. Under a fair-share scheduler, each workload gets an equal share of CPU, and the standard deviation is zero.

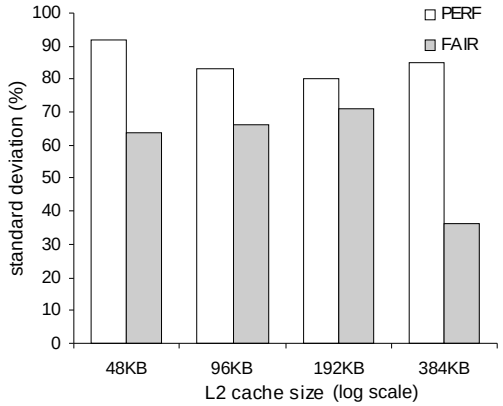

Figure 13. Evaluation of fairness for the two scheduling policies.

Figure 13 displays standard deviations for PERF and FAIR. Although when using the FAIR policy the scheduler is somewhat fairer than when using PERF, the standard deviation from the average is still rather high for the FAIR policy.

The level of fairness achieved in our experiments is not fundamental to the balance-set scheduling. It is particular to our workload. For example, if we had a workload where the number of cache-frugal threads was far greater than the number of cache-greedy threads, it would be possible to always match a cache-greedy thread with cache-frugal threads without sacrificing fairness. The relationship between the workload and the achievable fairness is a rich topic that we plan to investigate in the future.

The amount of fairness that is acceptable to sacrifice for the sake of performance is specific to each system. In systems where any fairness sacrifices would be unacceptable, balance set scheduling would not be the scheduling policy of choice. More likely, however, a system scheduler would want to be able to decide how much fairness it wants to trade off for performance, depending on relative priorities of the threads in its workload and on the performance goals set for the system. Therefore, it is important to equip the scheduler with mechanisms that would aid it in making such a decision. Our work on modeling effects of cache miss ratio on processor IPC [34] is a step in this direction.

5. IMPLEMENTING THE SCHEDULER

Implementing the balance-set scheduling algorithm is the subject of ongoing work. In this section we discuss the challenges involved in this task and lay out the related research agenda.

The scheduler operations that contribute the most to runtime overhead are the data collection for reuse-distance histograms and the miss ratio analysis.

Reuse-distance histograms need to be built when new threads enter the system and then re-built when the existing threads change their cache access patterns. Data collection for reuse-distance histograms can be done by monitoring memory locations accessed by a thread using the hardware watchpoint mechanism. Berg and Hagersten describe a user-level tool that builds reuse-distance histograms this way [19]. They report the runtime overhead of less than 20% for long-running applications. Handling kernel traps associated with watchpoints is the most significant source of overhead, most of which will be removed if the monitoring is done in the kernel. Only a sample of memory locations needs to be monitored. Reducing the sampling rate reduces the monitoring overhead, but can also result in reduced model accuracy. We plan to investigate what should be the right balance between the two.

The amount of storage needed for reuse-distance histograms is about 100 bytes per histogram. The range of possible reuse distances can be very large, so in order to reduce space requirements we compressed the histograms by aggregating reuse distances in buckets. Our predictions remained accurate even though we used fewer than 20 buckets.

Miss ratio analysis using the AVG method involves estimating miss ratios for individual threads and then averaging the quantities for groups of threads. These operations need to be performed as new threads enter or leave the system, as well as when reuse-distance histograms are updated for the existing threads. To avoid estimating miss ratios for all possible groups of threads, we plan to design a greedy algorithm that constructs a candidate set after analyzing the miss ratios for only a small number of thread groups. We plan to evaluate the cost of this process for different rates of arrival and departure of threads in the system.

When designing algorithms for multiprocessor operating systems, it is critical to avoid implementations that require global data, because this may result in shuffling such data among processors' L1 caches, resulting in high latencies [26,37]. We need to take this

important design principle into account when implementing the balance-set scheduler. On the other hand, it is possible, that this would not be so important on CMT processors, where the latency associated with faulting in the L1 is well hidden by hardware multithreading.

6. RELATED WORK

Previous work has proposed scheduling algorithms for single-core SMT processors that have been shown to improve system response time by 17% [1, 2, 12]. These algorithms involved sampling the space of possible schedules and using the ones that performed the best. This method can be implemented with virtually no overhead but requires hardware support. Our scheduling algorithm 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 equipped with dozens of thread contexts (i.e., a CMT processor) running hundreds of threads. 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.

We adapted the reuse-distance model to estimate miss ratios for threads concurrently accessing a cache. An alternative method has been proposed before [27]. Our methods are just as accurate but less computationally expensive.

A related study by Thekkath and Eggers considered co-scheduling threads based on their data-sharing patterns [28]. Although it is intuitive to expect that placing threads that share data on the same processor could improve performance in the L1 cache, the study showed that such a scheduling policy does not yield significant performance benefits.

Cohort scheduling [10] is a scheduling infrastructure for server applications that batches execution of similar operations from different requests. This 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.

The Capriccio thread package [17] implements resource-aware scheduling by monitoring threads' behavior, and measuring their resource requirements. This information is 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 rather detailed monitoring of the program's state.

7. CONCLUSIONS

In this paper we presented results of the first study evaluating the performance of a CMT processor. We analyzed how contention for the processor pipeline, L1 and L2 caches affects performance. We determined that contention for the L2 cache has the greatest effect on system performance – therefore, this is where system designers should focus their optimization efforts.

We investigated how to leverage the operating system scheduler to reduce the pressure on the L2 cache, using balance-set scheduling. To make balance-set scheduling work we adapted the reuse-distance cache model to estimate miss ratios for threads that concurrently access the cache.

We demonstrated that with balance-set scheduling it is possible to reduce the L2 cache miss ratio by 19-37% and increase performance by 27-45%. Performance improvement, however, may come at the expense of fairness.

To determine whether balance-set scheduling is viable for real systems, we plan to implement it and evaluate its runtime overhead. We also plan to investigate how workload characteristics affect the potential performance gains from this algorithm and the associated fairness tradeoffs.

8. 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] SPEC CPU2000 Web site: http://www.spec.org/cpu2000/analysis/memory/

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

[5] A. Agarwal, 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] Jonathan Schwartz on Sun's Niagara processor: http://blogs.sun.com/roller/page/jonathan/20040910#the_difference_between_humans_and

[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 Microsystems TR 2004-133*, March 2004.

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

[18] “IBM eServer iSeries Announcement”, http://www-1.ibm.com/servers/eserver/iseries/announce/

[19] E. Berg, E. Hagersten, “Efficient Data-Locality Analysis of Long-Running Applications,” TR 2004-021, University of Uppsala, May 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] F. W. Burton and M. R. Sleep, “Executing Functional Programs on a Virtual Tree of Processors”, *Proceedings of the ACM FPCA, Portsmouth, NH*, pp. 187-194, ACM, 1981.

[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 TPDS*, *14*, 6, pp. 581-592, 2003.

[34] A. Fedorova, M. Seltzer and M. Smith, “Modeling Effects of Memory Hierarchy Performance on the IPC of Multithreaded Processors”, *Harvard* University Technical Report, In Preparation. Contact fedorova@eecs.harvard.edu for a copy.

[35] M. Seltzer, P. Chen, J. Ousterhout, “Disk Scheduling Revisited”, *Proceedings of the USENIX Winter 1990 Technical Conference*

[36] K. Olukotun, B.Nayfeh, L. Hammond, K. Wilson and K. Chang, “The Case for a Single-Chip Multiprocessor”, *ASPLOS* 1996.

[37] 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

[38] D. Ellard and M. Seltzer, “NFS Tricks and Benchmarking Traps”, *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2003

[39] 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.