Chip Multithreading Systems Need a New Operating System Scheduler

Alexandra Fedorova*†, Christopher Small†, Daniel Nussbaum†, and Margo Seltzer*

*Harvard University, †Sun Microsystems

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 two years; understanding how such systems will perform is crucial if we are to use them to full advantage.

Our simulation experiments show that a CMT-savvy operating system scheduler could improve application performance by a factor of two. In this paper we describe our initial analysis of application performance on CMT systems and propose a design for a scheduler tailored for the needs of a CMT system.

1. INTRODUCTION

Modern server applications, such as application servers, web services, and on-line transaction processing systems, are notorious for poor utilization of CPU pipeline. Such applications usually consist of multiple threads of control, executing short stretches of integer operations, with frequent dynamic branches. This negatively affects cache locality and branch prediction and causes frequent processor stalls [1,2]. Modern superscalar processors, using speculative and out-of-order execution, typically manage to wring instruction-level parallelism (ILP) from scientific workloads, but can do little for transaction-processing-like 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-like 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 (as often as on each 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 systems. Intel's hyper-threaded Xeon is an MT system.

Driven by improvement in chip densities, hardware vendors are proposing architectures that combine CMP and MT. We will refer to such systems as *chip multithreading* (CMT) systems. Sun Microsystems, Intel, and IBM have announced plans to ship such systems as early as 2005 [8, 9,18]. Understanding what affects application performance on such systems is critical to our ability to best use such systems as they become available.

A major factor affecting performance on systems with multiple hardware contexts is operating system scheduling. For example, one OS scheduler that is tailored for MT processors produces an average performance improvement of 17% [10]. Moreover, it has been pointed out that a naïve scheduler can hurt performance, making a multithreaded processor perform worse than a single-threaded processor [17]. Our experiments have shown that the potential for performance gain from a specialized scheduler on CMT systems is even greater, and can be as large as a factor of two. We believe that scheduler designs proposed for single-processor multithreaded systems do not scale up to the dozens of hardware threads that we expect to find on proposed CMT processors. Such systems will require a fundamentally new design for the operating system scheduler.

An ideal scheduler will assign threads to processors in a way that minimizes resource contention and maximizes system throughput, and would do so in a scalable fashion. The scheduler must understand how its scheduling decisions will affect resource contention, because resource contention ultimately determines performance.

We undertook a simulation study to better understand the causes and effects of resource contention on CMT processors. In this paper we describe the results of our experiments, which have led us to a design for a CMT-savvy OS scheduler that improves application performance by as much as a factor of two.

The rest of the paper is organized as follows. In Section 2, we discuss related work and explain why existing scheduling algorithms will not work on CMT systems. We describe our system model and simulator in Section 3. In Section 4, we present our study of resource contention. In Section 5, we propose a design for a CMT-tailored scheduler. We conclude in Section 6.

2. RELATED WORK

Scheduling on single-processor MT systems has been studied before [10-12]. The scheduling algorithms for single-processor MT systems discussed in the literature worked as follows: they ran all combinations of threads that could be co-scheduled, determined which combination(s) yielded the best performance, using this data to make further scheduling decisions. These algorithms were shown to work reasonably well on single-process MT systems, yielding an average performance improvement of 17%. Although this technique is applicable on a system that supports a handful of threads, it does not scale well. An OLTP workload may involve a hundred threads; on a CMT system with 16 hardware contexts, there are 10^{27} combinations to evaluate. Moreover, while the scheduler is evaluating schedules, it is bound to try those that do not work well, and during these times the system is not running at best performance. We argue that there is a need for a different design. Our proposal involves building a scheduler that would model relative resource contention resulting from different potential schedules, and use this as a basis for its decisions.

We realize that modeling resource contention is a hard problem. In situations where a good prediction is difficult to achieve, we may resort to using our model of resource contention to reduce the size of the problem space, and then use the algorithms proposed in earlier work [10-12] to make final scheduling decisions.

3. EXPERIMENTAL PLATFORM

To study performance of these systems, 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 is configurable.

Our model of a multithreaded processor core is based on the concept of fine-grained multithreading proposed by Laudon et al. [6]. An MT core has multiple *hardware contexts* (usually one, 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 servicing a cache miss. When one or more threads are unavailable, the system continues to switch among the remaining available threads. For multithreaded workloads, this improves processor utilization and hides the latency of long operations. This latency-hiding property is at the heart of hardware multithreading.

Our simulated CPU core has a simple RISC pipeline, with one set of functional units (arithmetic unit, load/store unit, etc.). The processor that we model is typically configured with four hardware contexts per CPU core. Each core has a single shared TLB and L1 data and instruction caches. The (integrated) L2 cache is shared by all of the CPU cores on the chip.

One alternative to the architecture that we have chosen is to have multiple sets of functional units on each CPU core. This is termed a s*imultaneous multithreaded* (SMT) system [7, 14]. SMT systems are more complex and require more chip real estate. We have instead taken the approach of leveraging a simple, classical RISC core in order to allow space for more cores on each chip. This allows for a higher degree of multithreading in the system, resulting in higher throughput for multithreaded and multi-programmed workloads.

4. STUDYING RESOURCE CONTENTION

On a CMT system, each hardware context appears as a logical processor to the operating system; a software thread is assigned to a hardware context for the duration of the scheduling time slice. Threads that share a processor compete for resources. There are many different categories of resources contended for; in this paper, we focus on the processor pipeline because that category of contention characterizes the difference between single-threaded and multithreaded processors.

When assigning threads to hardware contexts, the scheduler has to decide which threads should be run on the same processor, and which threads should be run separately. The optimal thread assignment should result in high utilization of the processor. If we are to design a scheduler that can find good thread assignments, we must understand the causes and effects of contention among the threads that share a processor. Our study of such contention is the subject of this section.

We have observed that the instruction mix executed by a workload is an important factor in determining the level of contention for the processor pipeline. The key to understanding why this is the case is the concept of *instruction delay latency*, which we introduce next.

Recall from Section 3 that a processor keeps a copy of architectural state for each active hardware context. When a thread performs a long-latency operation, it is blocked; subsequent instructions to be issued by that thread are *delayed* until the operation completes. We term the duration of this delay the *instruction delay latency*. ALU instructions have 0 delay latency.[1] A load that hits in the L1 cache has a latency of four cycles. A branch delays the subsequent instruction by two cycles.

Processor pipeline contention depends on the latencies of the instructions that the workload executes. If a thread is running a workload dominated by instructions with long delay latencies, such as memory loads, it will often let functional units go unused, leaving ample opportunities for other threads to use them. Resource contention in this case is low. Alternatively, if a thread is running an instruction mix consisting strictly of ALU operations it can keep the pipeline busy at all times. The performance of other threads co-scheduled with this thread will suffer accordingly.

[1] We model a pipeline that implements result bypassing.

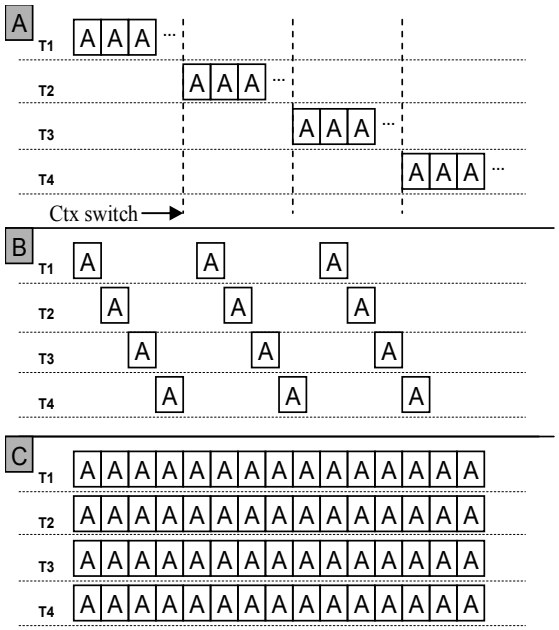

Figure 1. CPU-bound workload. Threads T1 through T4 contend for ALU on each cycle. Systems A and B perform comparably, because they each have one ALU. System C has an ALU per each of its four processors, and it outperforms A and B by a factor of four.

To assess the severity of pipeline contention for a workload, it is useful to compare how it would perform on an MT core with how it would perform on a conventional single-threaded core and on a traditional multiprocessor. Performance on the conventional processor provides the theoretical lower bound; performance on the multiprocessor provides the upper bound. When contention is low, performance on an MT core should approach that of an MP system, where each thread has all functional units to itself. When contention is high, performance of an MT core will be no better than that of a single-threaded processor, where a single thread monopolizes all resources.

Now let us show by example how this reasoning can apply in practice. We have two workloads: (1) a CPU-bound workload, consisting of four threads that execute only ALU instructions with delay latency of zero cycles, and (2) a memory-bound workload, consisting of four threads that execute only load instructions with delay latency of four cycles. We wish to see how these workloads would run on three systems: A, a single-threaded processor; B, a multithreaded processor with four hardware contexts; and C, a four-way multiprocessor.

When running the CPU-bound workload, A and B will perform comparably; C will have a throughput four times greater than the other two systems, because it has four times as many functional units. Figure 1 illustrates why this is the case.

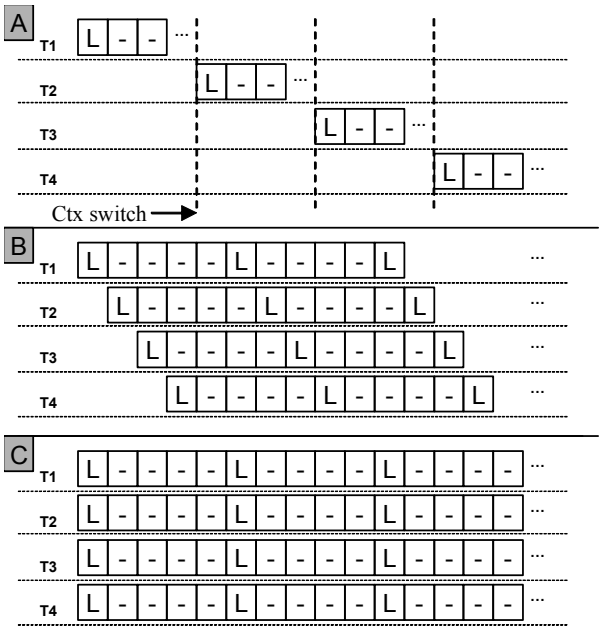

Figure 2. Memory-bound workload. Systems B and C outperform System A by a factor of four, because they are able to overlap memory access latencies for the four threads.

When running the memory-bound workload, System B and System C will perform comparably, outperforming System A by a factor of four[2]. Figure 2 illustrates this.

Figure 3 shows how an experiment performed on our simulator validates this theory. The instruction mixes executed by the threads have been hand-crafted to consist strictly of ALU instructions in the CPU-bound case, and strictly of load instructions in the memory-bound case.

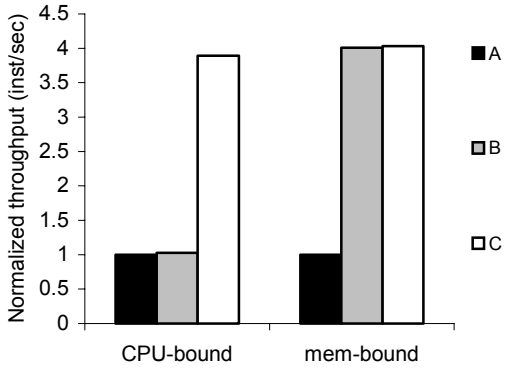

Figure 3. Combined throughput delivered by all threads in the system for each type of workload. Throughput has been normalized to the performance of System A.

[2]We assume that the cache hit rate and available memory bandwidth is no worse on B than on A or C.

This simple experiment demonstrates that the instruction mix, and, more precisely, the average instruction delay latency, can be used as a heuristic for approximating the processor pipeline requirements for a workload. The scheduler can use the information on the workload's instruction mix for its scheduling decisions. A thread with an instruction mix dominated by long-latency instructions can leave functional units underutilized. Therefore, it is logical to co-schedule it with a thread that is running a lot of short-latency instructions and has high demand for functional units.

We stress that this is a heuristic; since it does not model contention for other resources, it does not always predict performance. For example, this technique does not take into account effects of cache contention that surface when threads with large working sets are running on the same processor.

Consider, for example, the memory-bound workload, modified to vary the working set size of a thread from eight bytes to 2MB. Figure 4 shows the results of this experiment, comparing the three systems. According to our simple heuristic, B and C should perform comparably. However, in the middle part of the graph the two curves diverge. The cause is the reduced L1 data cache hit rate on B, where four threads share the cache on the processor.

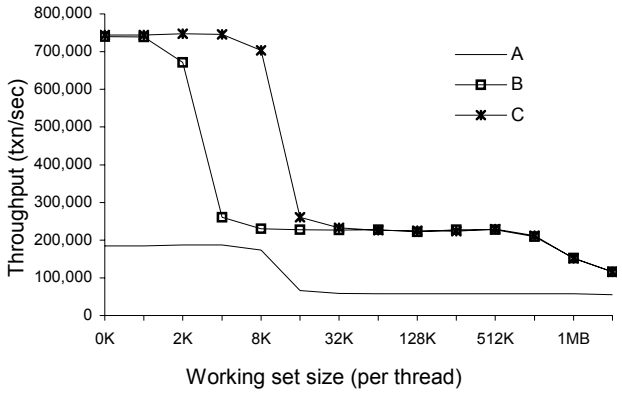

Figure 4. Memory-bound workload with varying working set size.

Although this model has limitations, we will show in the next section that it is a good starting point. We are currently extending our model to include effects of cache contention and cooperative data sharing.

5. SCHEDULING

In the previous section we argued that properties of the instruction mix can be useful in assessing pipeline contention and making scheduling decisions.

Mean cycles-per-instruction (CPI), easily determined using hardware counters, gives us a useful window into the dynamic instruction mix of a workload. CPI nicely captures average instruction delay, which can serve as a first approximation for making scheduling decisions.

In the following experiment, we make use of a thread's single-threaded CPI (the CPI that would be observed if the thread were running on a dedicated processor). Threads with high CPIs usually have low pipeline resource requirements, because they spend much of their time blocked on memory or executing long-latency instructions, leaving functional units unused. Threads with low CPIs have high resource requirements, as they spend little time stalled.

	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.

In this experiment we configured our simulated processor with four cores and four threads on each core. The workload consists of 16 threads, four each with CPI of one (CPU-bound), six, 11, and 16. We run four different schedules, assigning threads to cores as shown in Table 1.

We expect schedules (a) and (b) to perform better than schedules (c) and (d), because they schedule threads with low resource requirements (high CPI) together with threads that have high resource requirements (low CPI). In schedules (c) and (d), there are two processor cores running multiple threads with a single-threaded CPI of 1. Those cores would be fully utilized, while other cores would be under-utilized. Figure 5 shows the experimental results. Schedules (a) and (b) perform twice as well as (d) and 1.5 times better than (c).

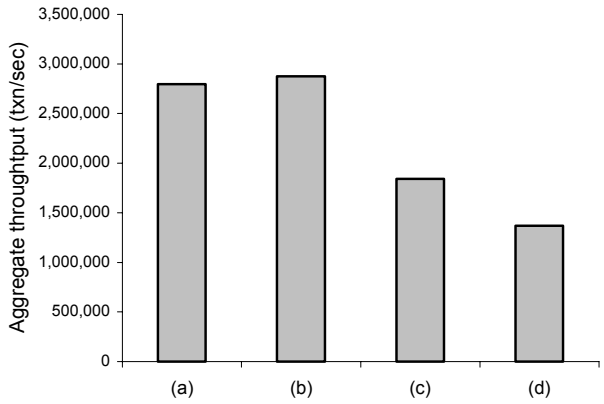

Figure 5. Aggregate throughput achieved by each schedule.

This begs the question as to how a scheduler can make these decisions in practice. We have measured the CPI of a number of applications and benchmarks over time, and found that (a) measured CPI can vary by an order of

magnitude among applications, and (b) given recent behavior, we can predict the near-term CPI of a thread. Given the range and predictability of CPI, constructing a scheduler that balances load across CPU cores on a CMT processor should be a simple matter of engineering. Due to space constraints, we do not elaborate any further on these results in this paper, but we plan to do so in future work.

6. SUMMARY

We have demonstrated that basing scheduling decisions on CPIs works well for simple workloads. However, we envision some limitations of this approach. CPI does not give precise information on the types of instructions that the workload is executing. For example, using just the CPI, the scheduler cannot tell which threads will compete for other resources, such as the cache hierarchy or a floating-point unit. Another disadvantage is that when the CPI of the workload is measured on a busy system, it may be affected by the resource contention that is already present.

We are currently investigating the following topics to develop better CMT schedulers:

- Techniques for inferring single-threaded CPI, given CMT CPI.
- Determining the effects of cache contention on the throughput of co-scheduled threads.
- Investigating other workload characteristics, e.g., static instruction mix, to improve scheduling decisions.
- Studying the nature and dynamics of CPIs exhibited by real workloads to understand whether this is a viable metric to be used for scheduling real applications.
- Investigating ways to integrate these ideas with other scheduling policies.
- Testing our scheduling ideas on real workloads.

In this paper we have demonstrated that CMT systems need new schedulers: a naïve scheduler may squander up to half of available application performance, and existing SMT scheduling algorithms do not scale to dozens of threads.

We have reported results of our simulation study that helped us better understand some of the causes and effects of pipeline contention on CMT processors. Based on our findings we proposed a scheduler that makes use of CPI. Our scheduler yields a two-fold performance improvement over a naïve scheduler.

7. ACKNOWLEDGMENTS

We would like to thank Tim Hill and Miriam Kadansky of Sun Microsystems for their useful suggestions and help with preparation of this paper.

8. REFERENCES

[1] James R. Larus and Michael Parkes, "Using Cohort Scheduling to Enhance Server Performance", *USENIX Tech. Conf.,* June 2002.

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

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

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

[5] Anant Agrawal, Beng-Hong Lim, David Kranz and John Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing", *ISCA,* June 1990.

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

[7] Jack Lo, Susan Eggers, Joel Emer, Henry Levy, Rebecca Stamm, and Dean 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] Allan Snavely and Dean Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Machine", In *ASPLOS IX*, November 2000.

[11] Allan Snavely, Dean Tullsen, and Geoff Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor", *SIGMETRICS*, 2002.

[12] Sujay Parekh, Susan Eggers, Henry Levy, Jack Lo, "Thread-sensitive Scheduling for SMT Processors", http://www.cs.washington.edu/research/smt/, 2000.

[13] Nathan Tuck and Dean M. Tullsen, "Initial Observations of the Simultaneous Multithreading Pentium 4 Processor", *PACT*, September 2003.

[14] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *ISCA*, June 1995.

[15] Peter S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner, **"**SimICS/sun4m: A Virtual Workstation", *USENIX Tech. Conf.*, June 1998.

[16] Daniel Nussbaum, Alexandra Fedorova, Christopher Small, "The Sam CMT Simulator Kit." Sun Labs TR. *In preparation; contact fedorova@eecs.harvard.edu.*

[17] Duc Vianney, "Hyperthreading Speed Linux", http://www-106.ibm.com/developerworks/linux/library/l-htl/

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