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

We investigated how operating system design should be adapted for multithreaded chip multiprocessors (CMT) – a new generation of processors that exploit thread-level parallelism to mask the memory latency in modern workloads. We determined that the L2 cache is a critical shared resource on CMT and that an insufficient amount of L2 cache can undermine the ability to hide memory latency on these processors. To use the L2 cache as efficiently as possible, we propose an L2-conscious scheduling algorithm and quantify its performance potential. Using this algorithm it is possible to reduce miss ratios in the L2 cache by 25-37% and improve processor throughput by 27-45%.

1. INTRODUCTION

This paper explores the subject of operating system design for multithreaded chip multiprocessors (CMT). CMT processors combine chip multiprocessing (CMP) and hardware multithreading (MT). 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 or some other long operation, other threads can make forward progress. Numerous studies have demonstrated the performance benefits of CMP and MT [1-4, 8, 15, 16, 20].

The hardware industry is turning to CMT as a way to improve future application performance because conventional techniques for hiding the latency of long operations, such as branch prediction and out-of-order execution, are failing to work for modern applications. These applications, such as web services, application servers, and on-line transaction processing systems, usually include multiple threads of control executing short sequences of integer operations, with frequent dynamic branches. This structure decreases cache locality and branch prediction accuracy and causes frequent processor stalls [7, 17, 18, 19]. Add to that the growing gap between processor and memory performance, and the result is very poor processor pipeline utilization: even relatively simple SPEC CPU benchmarks have pipeline utilizations as low as 19% [9]. This means that the majority of the time, the processor pipeline is unused. CMT processors address this problem. Most of the new processors released by IBM, Intel and Sun Microsystems today are MT, CMP or CMT [5, 6, 12].

CMTs can be equipped with dozens of simultaneously active thread contexts (e.g., Sun's Niagara processor will have eight cores, each with four thread contexts [5]), and, as a result the competition for shared resources is intense. Our approach is to identify the shared resource that is likely to become the performance bottleneck and then investigate operating system approaches for improving its efficiency.

In this paper we focus on processor caches[i]. We have found that the latency resulting from poor hit rates in the L1 cache can be effectively hidden by hardware multithreading, but that high contention for the L2 can significantly hurt overall processor performance. This result drove us to investigate how much potential there is in using OS scheduling to improve L2 (and subsequently overall processor) performance.

We have designed an OS scheduling algorithm based on the balance-set principle [14], and found that it has the potential to reduce the L2 cache miss ratios by 25-37%, 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.

2. PERFORMANCE BOTTLENECKS

In this section we show that while hardware multithreading does an excellent job of hiding latency resulting from L1 cache misses, its ability to hide memory latency resulting from L2 cache misses is limited.

We performed experiments using a CMT processor simulator built on top of Simics [10, 11]. We simulate a chip multiprocessor where each multithreaded core has a simple RISC pipeline supporting simultaneous execution of four threads, shared instruction and data caches (16KB and 8KB respectively), and a shared TLB. There is a unified shared L2 cache per chip, whose size we vary

depending on the experiment. For more details on our simulator and for the explanation of our choices for system configuration parameters, please refer to our earlier papers [11, 21].

To analyze processor sensitivity to the L1 cache miss ratio, we measured cache miss ratios and instructions per cycle (IPC) for the benchmarks from the SPEC CPU2000 suite, varying the size of the data cache from 8KB to 128KB. In 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.

Figure 1 shows the average miss ratios and IPC for the benchmarks. The key point of this figure is that even though the increase of cache miss ratios is significant (from 8% for the 128KB cache to 25% for the 8KB cache), the IPC is relatively insensitive to this variation.

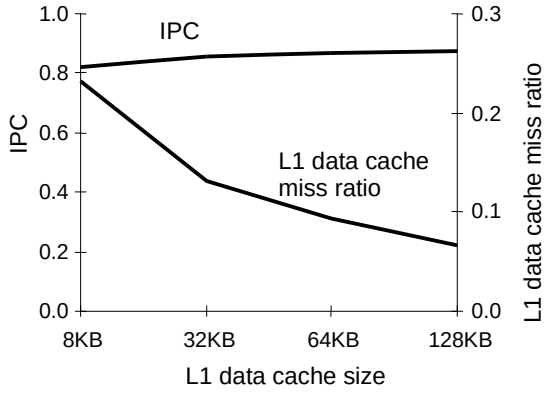

Figure 1. IPC and L1 data cache miss ratio for the SPEC workload

Now, consider a similar experiment with the L2 cache. We configured our simulator to have two cores, and ran the SPEC benchmarks simultaneously, creating a workload of heterogeneous threads.

Figure 2 shows the processor IPC and the L2 cache miss ratio for various L2 cache sizes. IPC degradation is evident as the L2 cache becomes smaller – this shows that the processor performance is significantly dependent upon the L2 cache.

This result is worthy of serious attention. Modern applications exhibit a trend of becoming progressively more memory-intensive. While CMT processors may be equipped with enough cache for the time being, it is likely that in the future it will be more difficult for CMT processors to satisfy application cache needs. Equipping these processors with larger L2 caches may be difficult: as the microchip technology is moving beyond the 90-nm mark, packing more and more transistors on a processor becomes increasingly more

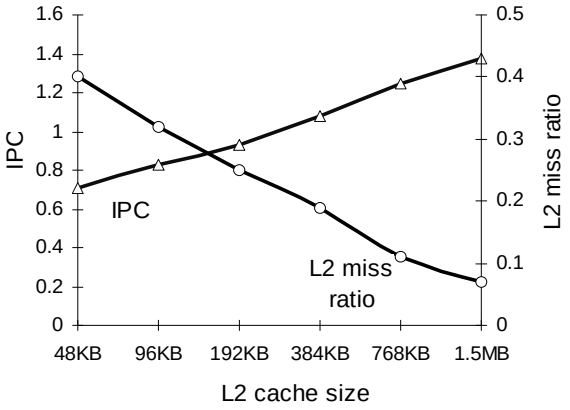

Figure 2. IPC and L2 miss ratio for the SPEC workload

complicated due to limitations of silicon technology. To ensure that our systems run well in the future, it is important that the software community develop techniques to improve resource utilization of CMT processors. In the next section we describe an L2-cache-conscious scheduling algorithm and quantify the potential performance improvement that it can produce.

3. BALANCE-SET SCHEDULING

Balance-set scheduling was proposed by Denning [14] as a way to improve 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 into 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 reduces cache miss ratios.

However, we found that working set size is not a good indicator of a workload's cache behavior [21]: the reuse pattern of memory locations in the working set is more important than the size of the working set. In order to estimate a cache miss ratio produced by a group of threads we used a cache model for single-threaded workloads developed by Berg and Hagersten [13], and adapted it to work for multi-threaded workloads [21].

Using this model we are able to estimate cache miss ratios of multithreaded workloads to within 17% of their actual values, on average. Such accuracy is satisfactory, because it is sufficient to distinguish between those workloads that fit in the cache and those that thrash.

Once we are able to estimate the cache miss ratio of any group of threads we can decide which threads should be scheduled together. By scheduling threads in groups that have low cache miss ratios, we ensure that

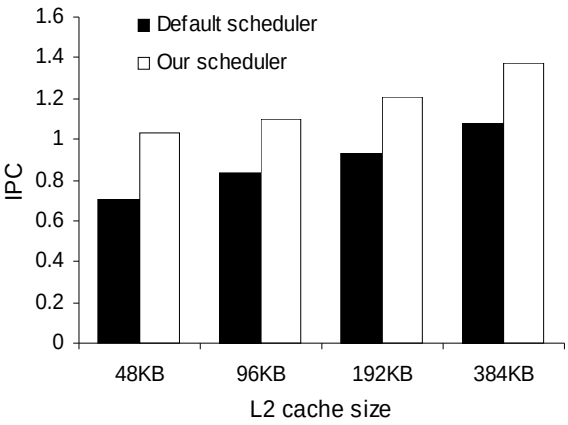

Figure 3. IPC achieved with the default scheduler and the balance-set scheduler.

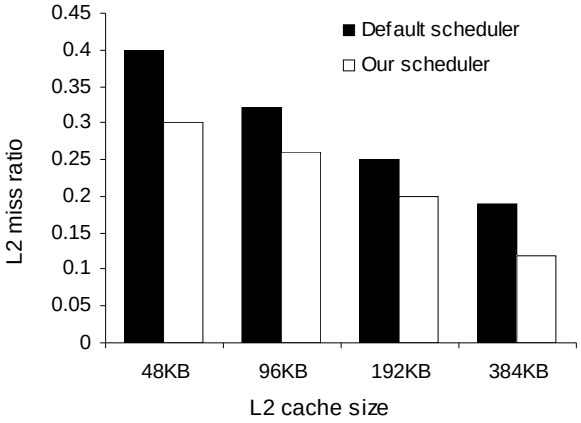

Figure 4. L2 cache miss ratios achieved with the default scheduler and the balance-set scheduler.

the miss ratio is always kept low, and the overall IPC is high. We also need to make sure that each runnable thread is included in at least one of the scheduled groups, so that none are starved.

The steps involved in our algorithm are:

1. Estimate cache miss ratios for all possible groups of threads using the Berg-Hagersten-based model.
2. Select cache miss ratio threshold
3. Schedule those thread groups whose estimated cache miss ratio is below the threshold.

Before implementing this algorithm in the operating system, we wanted to be sure that it would be worth the effort. Therefore, we quantified the potential for performance improvement using a scheduler prototype, which we implemented partially at user-level and partially inside the simulator. We now briefly describe the prototype implementation of these steps; a more detailed description appears in our earlier work [21].

Estimating cache miss ratios using the Berg-Hagersten model requires monitoring threads' memory re-use patterns. To implement such monitoring it is necessary to construct a sample of memory locations that a thread references and then to watch how often those locations are reused. Using this approach in a real system is expensive, because it requires handling frequent processor traps. Although we are working on a low-overhead way to approximate the measurements produced by the memory-monitoring approach, in this analysis we use the memory-monitoring approach because it provides the best accuracy. Our hardware simulator analyzes memory-reuse patterns of threads, and produces the data that we use to estimate cache-miss ratios for all groups of threads.

The cache miss ratio threshold guides the scheduling decisions. Only the thread groups whose estimated miss ratio is below the threshold will be scheduled. We select the cache-miss ratio threshold such that each thread is included in at least one of the groups that satisfies the threshold; of all such possible threshold values, we choose the lowest. This way we ensure that none of the threads are starved.

In our prototype we enforced scheduling decisions at user-level, binding threads to processors using the mechanisms available through a user-level library on the Solaris™ operating system.

We now present performance results achieved using our algorithm. We use the multi-process SPEC CPU workload – the same that we used for the analysis of IPC sensitivity in the previous section. We compare the IPC and the L2 cache miss ratio of this workload achieved using the default Solaris™ scheduler, and using our balance-set scheduler prototype. Figure 3 presents the IPC for both schedulers, Figure 4 presents the L2 cache miss ratio. In all cases, the balance-set scheduler outperforms the default scheduler. The IPC gain from the balance-set scheduler is from 27% (384KB L2) to 45% (48KB L2). This improvement in the IPC is due to reduction in cache miss ratios. As Figure 4 shows, with balance-set scheduling we were able to reduce the L2 miss ratios by 25-37%.

The performance improvement resulted from constructing thread groups so that they would share the cache amiably, producing a low L2 miss ratio, and achieving high IPC as a result.

4. DISCUSSION

The magnitude of the performance improvement from balance-set scheduling depends on the properties of the workload. Since the balance-set scheduler exploits the disparity in threads' cache behaviors, if all threads in the workload behave in a similar manner,

large performance gains cannot be realized. Understanding how the characteristics of the workload affect potential performance gains is the subject of future work. We also need to determine how balance-set scheduling affects fairness: sacrificing fairness for performance is a canonical trade-off made in scheduling algorithms.

Implementing cache-miss ratio analysis using memory monitoring and examining all possible groups of threads to select those that satisfy the miss ratio threshold is expensive. We are working on a novel low-overhead way of approximating these operations and are now implementing the balance-set scheduler in Solaris™ 10.

5. CONCLUSIONS

In this paper we presented results of the first study evaluating the performance of a CMT processor. We analyzed how contention for 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.

We demonstrated that with balance-set scheduling it is possible to reduce the L2 cache miss ratio by 25-37% and increase performance by 27-45%.

The implementation of the scheduler is currently under way. In the future, we also plan to investigate how workload characteristics affect the potential performance gains from this algorithm and the associated fairness tradeoffs.

6. ACKNOWLEDGEMENTS

The authors thank Tim Hill, John Davis and James Laudon of Sun Microsystems, and Eric Berg of University of Uppsala for help with preparation of this paper.

7. REFERENCES

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

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

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

[4] 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.

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

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

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

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

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

[10] P. Magnusson et al. SimICS/sun4m: A Virtual Workstation. *USENIX*, June 1998.

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

[12] IBM eServer iSeries Announcement. http://www-1.ibm.com/servers/eserver/iseries/announce/

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

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

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

[16] 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.

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

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

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

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

[21] 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.

[i] We found previously that other shared resources have a smaller potential of becoming performance bottlenecks [21].