Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters *

Stephen Ziemba, Gautam Upadhyaya, and Vijay S. Pai
Purdue University
West Lafayette, IN 47907
sziemba@purdue.edu, gupadhya@purdue.edu, vpai@purdue.edu

Abstract

This paper analyzes the impact of scheduling decisions on dynamic task performance. Performance behavior is analyzed utilizing support workloads from SPECWeb 2005 on a multicore hardware platform with an Apache web server. Hardware performance counter data is collected via extending the Linux scheduler and analysis is then performed by core, by task, and by various metrics. The results show that considering a single per-core metric (such as IPC or cache miss rate) is not sufficient to categorize application behavior, since different thread types often have highly varying characteristics. Additionally, threads behave differently based on what thread was scheduled beforehand (seeing as much as 50% performance degradation when HTTP processing threads are preceded by long-running dynamic content PHP threads) or based on the length of their time slices (with longer-running PHP threads achieving 3 times the IPC of short-running ones on average).

1 Introduction

Multicore processors are now universal. The number of cores is experiencing exponential growth with only slow increases in frequency. Consequently, future increases in productivity and performance will arise primarily through exploiting parallelism with more aggressively multithreaded applications.

As with previous generations of general-purpose machines, all resources in a multicore chip are controlled by the operating system. In general, operating system design for multiprocessors is complicated by issues such as load-balancing and locality [17, 18, 19]. Multicore chips further introduce issues related to resource sharing [6]. As the number of threads increase, inter-thread communication becomes more significant; even some servers today have substantial inter-thread communication for purposes such as dynamic content generation.

The OS scheduler is particularly critical to the performance of multicore systems. It is responsible for balancing load across cores, selecting which thread to run on each core at any given time, and deciding how much time to give a running thread before preempting it. Scheduling decisions are fairly straightforward when only one application runs with just one long-running compute-bound thread per core, but become much more complex when multiple threads are used on each core to tolerate I/O latencies, when different types of threads communicate to yield a final result, or when the workload of each thread is highly variable over time.

Optimal scheduling of multiprocessor systems in general is an NP-hard problem; even an extremely restricted case such as offline load-balancing of independent tasks with known runtimes on two processors is equivalent to the partition problem, which is NP-complete [11]. Consequently, schedulers use heuristics to guide their decisions, based on notions of load balance and task priority. However, these heuristics can lead to poor performance for workloads that do not match their assumptions. This paper analyzes the impact of scheduling decisions on the dynamic performance of tasks for the SPECWeb 2005 benchmark. The workloads are run on an Apache web server, which uses a combination of threads, events, and processes to achieve concurrency. The Linux scheduler is instrumented to collect data from the hardware performance counters, which track dynamic events at each core such as retired instructions or L2 cache misses [2, 7]. The paper then analyzes the data by core, by task, and by various thread schedulings and time slicings to show which counters matter the most and which sorts of schedules can lead to degraded performance.

2 Background

This work builds upon the existing Linux scheduler and relates to other research results on scheduling. This section provides the background needed to understand this paper. Section 5 covers other related work.

Terminology. Various systems use different terminology for tasks commonly associated with the scheduler. This paper will consistently use the POSIX terminology for processes and threads: a process is a single running executable with a single memory space and one or more threads; each thread has its own PC, register set, and stack. The scheduler operates on threads. (Linux internally uses the term "thread group" for

*This work is supported in part by the National Science Foundation under Grant No. CCF-0532448.

the former and "process" for the latter, but most systems and most application programmers use the POSIX terms.)

In a multicore environment the scheduler is responsible for partitioning threads among individual cores, for selecting which of the runnable threads on a core should actually be executed at any given time, and for selecting how much time to allow that thread to run before forcing a preemption. We will refer to these components as load-balancing, task selection, and time-slicing, respectively. All are integral components of scheduling.

Current Linux scheduler. The most commonly used and stable scheduler for the Linux operating system is the *O(1) scheduler*. This process scheduler has been used since Linux version 2.6 [10]. A new scheduler, called the Completely Fair Scheduler (CFS), was introduced into the Linux kernel in version 2.6.23. CFS enforces fairness by using a time-ordered red-black tree to create an execution timeline. However, scheduling for fairness does not necessarily yield the best performance [4]. Consequently, CFS has been targeted primarily for interactive (typically desktop) environments rather than servers. Our workloads of interest are server-based, so the O(1) scheduler is used as our base. Other schedulers target specific environments such as real-time or embedded systems and are not considered further here.

The O(1) scheduler includes per-core structures called *run queues*, which contain the set of runnable threads assigned to that core. A run queue is split into two different arrays: the *active* array for threads that have not yet used their full timeslices, and the *expired* array for threads that have. Every task has a static priority defined at task creation. Static priority is modified positively or negatively based on the amount of time the task sleeps versus runs. Priority is recalculated at timer ticks, when a thread wakes up, or if a priority recalculation function is called. A task which spends more time sleeping is typically I/O bound and will see an improved priority, since such tasks are likely to need quick service when they awaken again. A task that uses most of its time slice running is typically CPU bound and will see its priority degrade. The time slice given to a task is a function of the priority. If a task does not use its entire time slice or if it runs too long while a runnable task of the same priority is waiting, it is put back on the active array at the end of the list rather than being moved to the expired array. Within a core, tasks are selected in priority order, with FIFO ordering among tasks with the same priority. Tasks in the expired array will typically have less immediate need for CPU time than those in the active array since they have already consumed their full time slice. Tasks in the expired array also should not run to provide tasks in the active array a fair share of CPU. When the active array becomes empty the active and expired arrays are swapped and the process repeats.

The O(1) scheduler includes load balancing to efficiently utilize all available cores. The scheduler generally follows affinity scheduling, in which threads prefer to remain on the same core in order to improve cache locality [17, 18, 19]. Every 200 ms, load balancing is triggered on a core to steal tasks from the busiest core to bring the cores into closer balance. Some tasks cannot be migrated, either because they are not runnable on all cores, or because they have recently run and are still considered to have useful data in the cache at their core (known as *cache hot* status). However, threads can be forced to move regardless of cache hot behavior if load balancing fails a sufficient number of times. Further, a core invokes *idle balancing* at any time before going idle, thus trying to find work for it to perform even though its own queues are empty. Like load balancing, idle balancing attempts to steal tasks from the currently busiest core to bring the run queues into a rough balance.

Decisions regarding load balancing, task selection, and time slicing are all based on heuristics regarding system load, expected system performance, and past task behavior. Such heuristics use minimal information to facilitate quick decision-making in the scheduler, though it is possible that using more detailed information would allow the actual application of interest to achieve higher performance.

Performance counters. All modern microprocessors expose information about their dynamic performance through performance counters. Typical performance counters include information such as the number of instructions retired, the number of cache misses at each level, the number of mispredictions, and the number of interrupts. Despite having a large number of events that may be monitored, typical processors only allow for a small number of performance counters to be tracked at a time [2, 7]. Tools such as Perfmon, HPCView, and VTUNE provide convenient interfaces to these performance counters [3, 13, 8].

In general, performance effects are difficult to attribute to specific instructions in processors with out-of-order instruction issue. The performance counters themselves are often speculative. Even effects such as cache misses do not increase execution time if they can be overlapped fully behind other cache misses or computation [14, 15]. Similarly, IPC may not be a meaningful measure of performance in multiprocessor codes if they use spin-locks or simple busy loops. Nevertheless, metrics such as IPC and L2 cache misses are typically correlated to application performance.

Research results targeted toward multicores. Fedorova et al. promote performance isolation of threads running on multicore processors by increasing CPU timeslices whenever threads in the system affect each others' cache performance [6]. In an earlier work, the above authors use performance counter information related to L2 cache miss rate combined with an analytical model to guide an SMT scheduling policy that leaves hardware contexts idle when using all of them would hurt performance by thrashing the L2 cache [5]. Each of the above works uses a synthetic workload consisting of multiple single-threaded applications. Using a standard benchmark workload would also help to characterize real behaviors that can arise in server systems (e.g., dynamic content generation, filesystem access, and network I/O). Additionally,

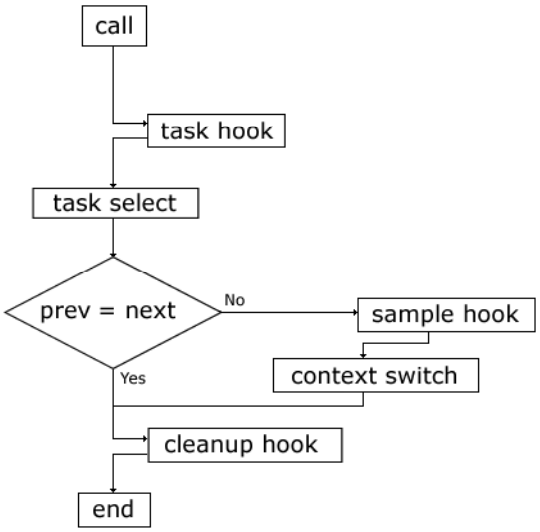

Figure 1. Instrumentation hooks added to Linux scheduler.

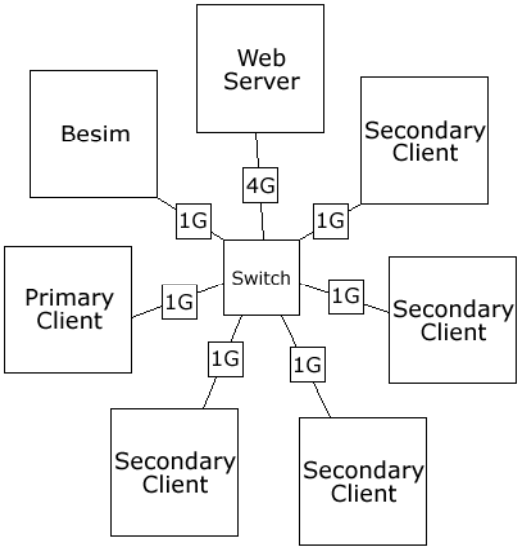

Figure 2. Experimental testbed configuration.

investigating multi-threaded and communicating applications may expose additional ways in which scheduling may impact performance since tasks can use shared resources constructively rather than just destructively.

3 Tracking the Effectiveness of a Multicore Scheduler

This paper gauges the effectiveness of scheduling policies by using feedback from performance counters. This requires instrumenting the Linux scheduler to collect and store performance counter information. Although there are tools available to provide access to and configure the performance counters, nothing fit our requirements for fast, flexible, and non-intrusive data tracking. Our utility operates at the kernel level with very little overhead. Interaction with userspace is avoided except for debugging purposes or for exporting data for offline analysis.

Scheduler instrumentation. The scheduler is extended to take performance counter samples at each context switch using the `rdmsrl` and `wrmsrl` instructions (read/write model-specific register). This approach is lightweight and allows precise per-thread samples to be collected. Other tools provide strategies such as monitoring time intervals or counter overflows (which is impractical in kernel space). The cores used in this study can monitor up to four separate performance counters simultaneously. The counters are set to monitor retired instructions, L2 cache data misses, L2 cache instruction misses, and L2 cache data accesses.

Figure 1 shows an abbreviated flow diagram of the scheduler. The instrumentation code is inserted via function pointers (which add minimal overhead) which can be enabled, disabled, and modified with ease. The majority of the instrumentation code is within the *sample_hook* located just before the existing `switch_to` function within `context_switch`. Due to the location of this hook, this code is run after the scheduler has selected the next thread. The `switch_to` function is only called if the current thread is different than the previous, so no sample is taken unless a context switch actually occurs. If desired, the *task_hook* could be configured to count this event. In the *sample_hook*, the module collects data from the performance counters, the threads process group and the processor's standard cycle counter on every context switch. The hook can also collect, calculate and store at desired intervals global statistics such as per-core IPC and miss rates. This information is then appended to the end of a circular array added to the task structure of each thread. The above information can optionally be exported to a large buffer within the module itself for later export and analysis. This sampling adds only very minor overhead to every context switch. The "task hook" and "cleanup hook" shown are responsible for setting up, storing and cleaning up values necessary for the proper operation of the *sample_hook*. Some values must be calculated outside of *sample_hook* due to the unlikely possibility that the schedule function must loop.

Experimental Methodology. The instrumented scheduler is tested using the Apache web server version 2.2.6 running the SPECWeb 2005 "Support" benchmark as an experimental workload. The server is compiled with all modules and shared modules, in addition to cgi and ssl support. External PHP threads are used for dynamic content generation. (These are called PHP threads throughout the results, whereas the Apache threads are called HTTP threads.) The tests are run on SunFire v40z systems that include 2 dual-core AMD Opteron processors (four core total) with 4 GB of RAM and 8 identical SATA hard drives. Web server data is distributed evenly

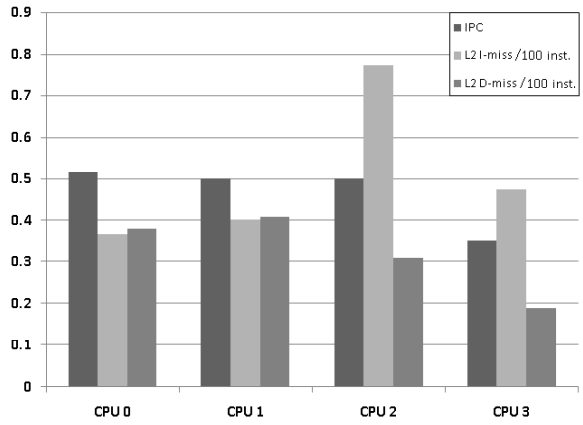

Figure 3. Average per CPU performance of Support run.

across 8 identical drives, and one drive also contains the PHP and operating system files.

The SPECWeb benchmark is run on seven different machines connected via Gigabit Ethernet to a switch. The machine types can be broken into four categories: the web server, the primary client, the back-end simulator (Besim), and the secondary clients. The web server runs on its own machine with four 1 Gbps NICs. These interfaces are on separate subnets in order to control network communication and promote balancing of network traffic with the secondary clients. The primary client runs on its own machine with one 1 Gbps NIC. The Besim runs on an Apache web server of its own with one 1 Gbps NIC. The remaining four machines run as secondary clients and each get their own 1 Gbps NIC. These machines are each put on a different subnet so that communication is distributed between the four Ethernet ports of the web server. All SPECWeb tests are run using 3350 client connections. SPECWeb reports the percentage of connections that meet "good" and "tolerable" quality of service constraints.

4 Results and Discussion

Figure 3 shows the per CPU performance for an execution of the SPECWeb 2005 support benchmark. Each set of bars represents performance counter measures from a different core. Within a set, the first bar represents the retired instructions per cycle (IPC); the second shows the number of L2 cache misses caused by instruction fetching for every 100 retired instructions (I-miss rate[1]); and the third shows the number of L2 cache misses caused by data operations for every 100 retired instructions (D-miss rate). CPU 3 has the poorest IPC of these cores, yet it has among the best I-miss and D-miss rates. This result seems counter-intuitive since IPC normally should be correlated with miss rate and the CPUs should perform similarly. The cache behavior of CPU 2 is quite different from that of CPU 0 or CPU 1, yet the IPC performance is roughly the same. Figure 4 shows the percentage of threads of each type (idle, OS, PHP, and HTTP) scheduled on each of the cores. There is no apparent direct link between thread distribution on a core and the core's performance. CPU 0 schedules the most PHP threads and the fewest HTTP threads, while CPU 3 schedules the fewest PHP threads (a factor of 3 fewer than CPU 0) and the most HTTP threads. CPUs 1 and 2 schedule more HTTP threads than CPU 0 yet have very similar IPC to CPU 0. This shows that while overall core performance is a good method of benchmarking core performance, it provides little insight to why the core behaves the way it does.

Figure 5 shows the break down of execution time spent at each core core on the various types of tasks. All of the cores except for CPU 3 spend a majority of execution time on PHP tasks. Although CPU 3 spends less time on PHP than CPU 0, the difference is less than a factor of 2 despite the fact that the number of PHP threads scheduled is lower by a factor of 3. Based on the execution time breakdown of CPU 3, PHP tasks on average must run twice as long on CPU 3 as they do on any other CPU. It is not likely that this discrepancy is the result of individual PHP tasks behaving differently, since 95% of PHP tasks are scheduled at least once on each CPU and 99% of PHP tasks are scheduled on at least 3 CPUs. (As discussed in Section 2, migration takes place despite a preference for affinity.) However, HTTP threads migrate much less frequently across all CPUs with only 37% of all HTTP threads ever being scheduled on CPU 0, compared to 59% for CPU 3. This uneven distribution of processes makes for an inconsistent environment for threads to execute on when they are migrated. The average execution length of HTTP threads is short enough for the environment to have a notable impact on their performance at any CPU. However, given the average length of a PHP thread on CPU 3 (4.4 ms), the environment can only impact the thread for a short time before the previous cache state becomes irrelevant (the task "warms up"). The results seem counter-intuitive since the environment differences seen in Figure 5 are in no way correlated to the IPC/cache performance seen in Figure 3.

Figure 6 details the performance of the two most heavily scheduled PHP tasks as they migrate across cores. In addition to the 3 previous performance counter bars, each set of bars also includes information about the average run time for that task on that core (with a Y-axis scale on the right hand side). The threads' behavior across cores is sporadic with average run time varying by as much as a factor of 19. These threads also exhibit much poorer IPCs on CPUs 1 and 2 than on the other cores, an effect which was not visible in the per-core aggregate statistics shown in Figure 3. This effect seems correlated to the short runtimes when these threads are scheduled on these cores.

Figure 7 breaks down the threads scheduled on CPU 0 in ranges based on the mean (μ) and standard deviation (σ) of

[1] Although the term miss rate is commonly used to refer to the percentage of cache accesses that miss (more properly termed *miss ratio*), we normalize misses by retired instructions so as to get a more direct measure of the impact of those misses.

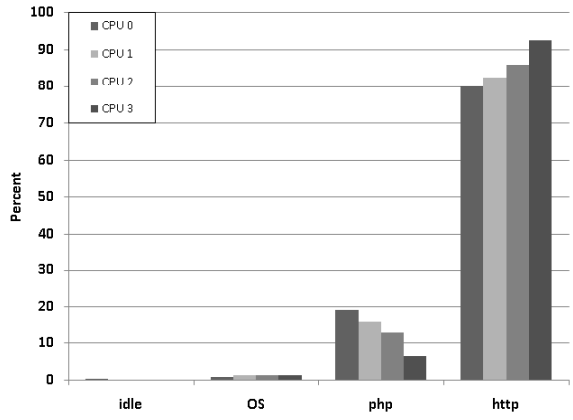

Figure 4. Schedules by thread type on each CPU.

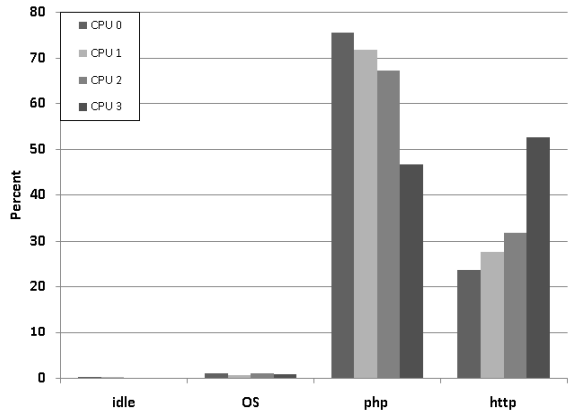

Figure 5. CPU time by thread type on each CPU.

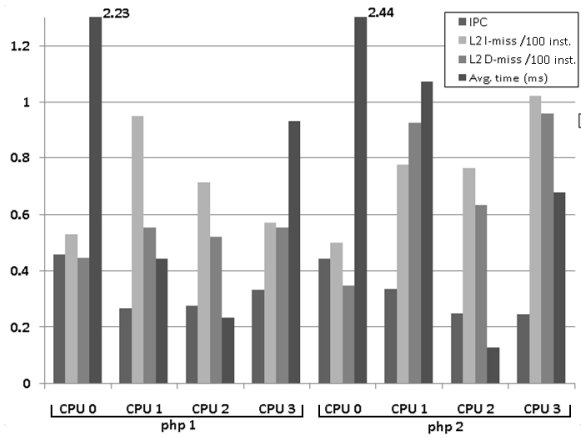

Figure 6. Performance breakdown of most commonly scheduled PHP threads.

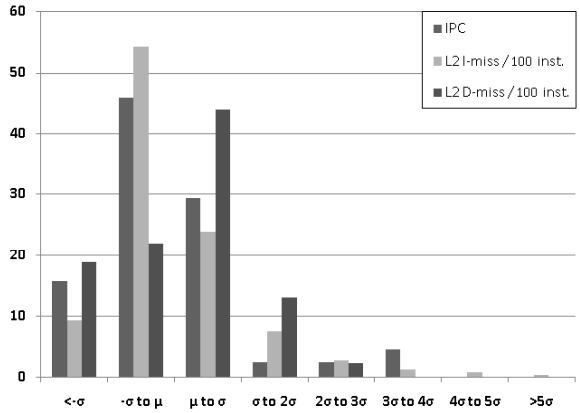

Figure 7. Categorization of threads on CPU 0 according to the percentage in each range based on the mean (μ) and standard deviation (σ) of each performance counter metric.

each performance counter metric. For example, the IPC bar for the range labeled μ to σ shows the percentage of threads for which the mean IPC is at least the mean IPC across all threads but no more than one standard deviation higher. While a significant percentage of the tasks fall within a single deviation of the mean IPC, a large fraction (6.8%) of the tasks have an IPC more than 2 standard deviations greater than the mean. More tasks fall 2 deviations away than 1 deviation away, and 80% more tasks fall 3 deviations away than 2. This behavior results in a graph which is slightly bimodal. HTTP threads account for 86% of schedules yet only get 34% of CPU time. Since there are much more HTTP schedules, the PHP contribution to the graph is minimized. Since no correlation exists between number of schedules and CPU time it is necessary to analyze process groups separately in order to understand the behavior of either process group.

Figure 8 breaks down HTTP performance on CPU 0 based on which thread or threads ran previously. HTTP tasks scheduled immediately after a different HTTP task on average had 7% better IPC, 7.5% better L2 I-miss rate and 60% better L2 D-miss rate. IPC and L2 miss rate continue improving as more sequential HTTP threads schedule until IPC improves 10.5% versus the average of all HTTP threads. Figure 9 shows the frequency of the events in Figure 8. About 10% of HTTP threads are preceded by long-running PHP threads. These HTTP threads get only 45% of the IPC and thus take twice as long to complete than an average HTTP thread. When compared to the best case, these threads get 41% of the IPC and take 2.4 times as long to complete than if they had been scheduled more effectively. All in all, about 10% of overall CPU time is wasted because of an HTTP thread being scheduled after a long-running PHP thread.

Figure 10 shows the performance of PHP threads with respect to their average run time on CPU 0. PHP tasks which run for less than 1 ms get on average 52% of the average IPC of all PHP tasks. By the time a PHP task has run for 4 ms, the

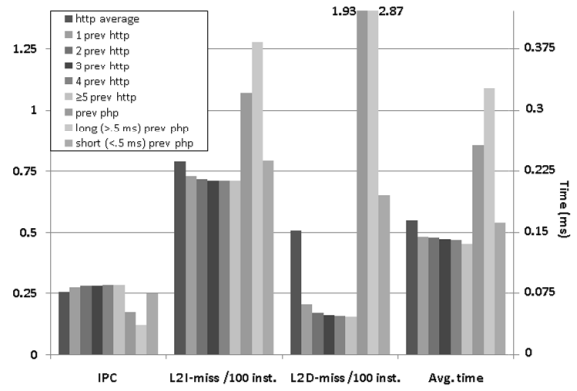

Figure 8. HTTP thread performance on CPU 0 based on previous thread(s) scheduled.

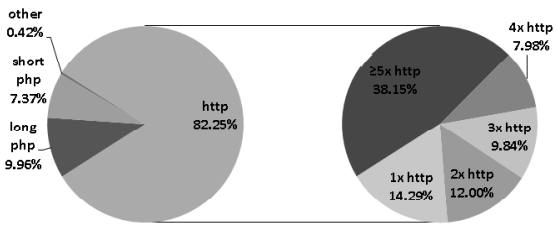

Figure 9. Frequency of thread schedules that affect HTTP thread performance on CPU 0.

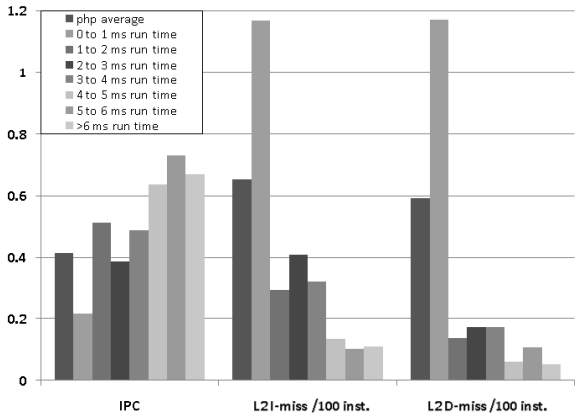

Figure 10. PHP thread performance on CPU 0 based on thread run time.

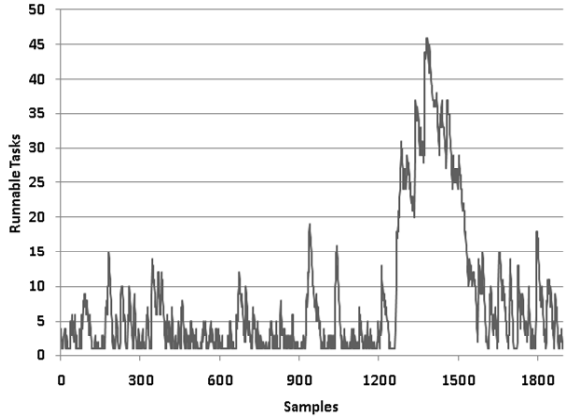

Figure 11. Number of runnable threads on HTTP CPU when HTTP tasks are given their own CPU. Samples taken every context switch.

IPC has improved to 154% of average IPC. IPC on CPU 0 can get as high as 176% of average IPC or 334% better than that seen when the thread was first scheduled. Scheduling PHP tasks for a short amount of time is inefficient, and interrupting long-running tasks will substantially harm overall work per unit of CPU time (despite being potentially more fair).

Discussion. The results regarding the IPC of HTTP threads indicate that it might be beneficial to segregate HTTP and PHP threads. Since HTTP threads need far less CPU time, we ran an experiment in which HTTP threads were dispatched to only a single CPU and PHP to the remaining 3. The results showed a slight drop in performance, with the number of good and tolerable connections, as well as the average response time, all degrading by 3–6%. The HTTP CPU spends 26% to 30% of the time idle while the PHP CPUs averaged around 4% idle. Figure 11 shows how saturated the run queue can get due to bursty request behavior. The worst observed case was 48 tasks waiting for the HTTP CPU. Assuming best case execution of .14 ms per thread, it would take 7 ms to empty the queue assuming no new threads are queued. This in turn increases the response time by 3-6% when average response time is roughly 6 seconds. Although this test sees a slight degradation in performance caused by excess queuing, the amount of CPU time needed for HTTP threads drops by nearly 47%. Mixing HTTP and PHP threads reduces effective work per unit of CPU time, and it can actually thus help overall CPU effectiveness to leave a CPU idle rather than filling its gaps with the non-preferred task type.

As a further extension, it would be valuable to have the scheduler make policy decisions based on the performance counters. As shown in Figure 10, the scheduler may want to make decisions based on how the performance counter values evolve as time slice increases. Such tracking can be added by, for example, instrumenting the timer tick code. Although an online decision-making process would have to be more responsive than offline analysis, techniques such as moving averages and histograms could still be used to get approximate information quickly. However, policy selection would still be

difficult; as shown in Figure 6, the performance of a thread even on any specific metric depends heavily on the environment at a core, making counter-based dynamic load balancing of individual tasks unlikely to succeed.

5 Related Work

Sections 1 and 2 covered the works most closely related to the development of this paper. This section discusses other theoretical and experimental approaches to scheduling problems.

The use of run-time measurements and there potential for improving task performance is not new. Before performance counters became available simpler metrics have been utilized to guide scheduling decisions [16]. Classification of I/O bound and interactive threads and control of execution time yielded positive throughput improvements.

Although most theoretical works on scheduling focus on cases where each task requires a single processing element, some consider cases such as "malleable processes" that may run on one or more cores with time dependent on the number of allocated processes [9]. Although such an approach does not directly account for communication, its impact is implicit through the use of non-linear speedup with the number of cores. Such works use heuristics known for other NP-complete or NP-hard problems such as strip packing to derive approximate schedules. Practical scheduling problems must also consider issues such as a dynamic number of threads associated with a task (e.g., PHP or Apache threads being started and stopped as connections arrive and complete).

Other works have considered more fine-grained problems, such as the scheduling of loop iterations in a parallel program. Lo et al. show that loop iteration scheduling should prefer cyclic distribution rather than block distribution in an SMT processor, thus encouraging fine-grained cache sharing [12]. Xue et al. describe a strategy for locality-aware loop iteration distribution in a multicore processor, using a static partition combined with dynamic load balancing [20]. Chen et al. show that a new scheduling algorithm called Parallel Depth First exceeds the performance of traditional Work-Stealing policies when executing benchmarks with fine-grained partitioning on a multicore processor with a shared cache [1]. The strategies used in this paper have many similarities with the latter works targeting multicores, but the domains are fundamentally different: this paper focuses on server applications where work arrives dynamically with new connections and the threads are a mix of I/O-bound network service threads and computation-bound PHP threads.

6 Conclusions

This paper uses performance counters as a tool to analyze the impact of scheduling decisions for a multicore system running a heavily multithreaded dynamic-content web workload. The results show that considering a single per-core metric (such as just IPC or cache miss rate) is not sufficient to categorize application behavior, since different types of threads often have highly different characteristics. Additionally, threads behave differently based on what other threads are scheduled beforehand or based on the length of their time slices. Such observations suggest that under some circumstances, it may be advisable to segregate thread types even if that ends up creating some CPU idle time. Although this paper only focuses on analysis, the results indicate that using performance counters and thread-specific information in making scheduling decisions could help to improve CPU efficiency and application performance.

References

[1] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 105–115, June 2007.

[2] P. J. Drongowski. Basic Performance Measurements for AMD Athlon^{TM} 64 and AMD Opteron^{TM} Processors. *AMD Developer Central*, December 2006.

[3] S. Eranian. The perfmon2 interface specification. Technical Report HPL-2004-200(R.1), HP Laboratories, 2005.

[4] A. Fedorova, M. Seltzer, C.Small, and D. Nussbaum. Throughput-oriented scheduling on chip multithreading systems. Technical Report TR-17-04, Division of Engineering and Applied Sciences, Harvard University, August 2004.

[5] A. Fedorova, M. Seltzer, and M. D. Smith. A Non-Work-Conserving Operating System Scheduler for SMT Processors. In *Proceedings of the 2nd Annual Workshop on the Interaction Between Operating Systems and Computer Architecture (WIOSCA)*, pages 10–17, June 2006.

[6] A. Fedorova, M. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, April 2007.

[7] Intel Corporation. *Intel$^{(R)}$ 64 and IA-32 Architectures Optimization Reference Manual*. 2007.

[8] Intel Corporation. *VTune(TM) Performance Environment User's Guide*. Number 310866-001. 2007.

[9] K. Jansen. Scheduling Malleable Parallel Tasks: An Asymptotic Fully Polynomial Time Approximation Scheme. *Algorithmica*, 39(1):59–81, January 2004.

[10] M. T. Jones. Inside the Linux scheduler. *IBM DeveloperWorks*, June 2006.

[11] R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[12] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parek, and D. M. Tullsen. Tuning Compiler Optimizations for Simultaneous Multithreading. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 114–124, December 1997.

[13] J. Mellor-Crummey, R. Fowler, and G. Marin. HPCView: A Tool for Top-down Analysis of Node Performance. In *Proceedings of the Second Annual Los Alamos Computer Science Institute Symposium*, October 2001.

[14] K. Oner and M. Dubois. Effects of Memory Latencies on Non-Blocking Processor/Cache Architectures. In *Proceedings of the International Conference on Supercomputing*, 1993.

[15] V. S. Pai, P. Ranganathan, and S. V. Adve. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

[16] F. Silva and I. Scherson. In *Improving Parallel Job Scheduling Using Runtime Measurements. In Proceedings of the 6th Workshop on Job Scheduling Strategies for Parallel Processing*, 2000.

[17] M. Squillante and E. Lazowska. Using Processor-Cache Affinity in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[18] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, February 1995.

[19] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling on Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 26–40, October 1991.

[20] L. Xue, M. Kandemir, G. Chen, F. Li, O. Ozturk, R. Ramanarayanan, and B. Vaidyanathan. Locality-Aware Distributed Loop Scheduling for Chip Multiprocessors. In *Proceedings of the 20th International Conference on VLSI Design*, pages 251–258, January 2007.