Interference Aware Cache Designs for Operating System Execution

David Nellans, Rajeev Balasubramonian, Erik Brunvand

UUCS-09-002

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

February 3, 2009

Abstract

Large-scale chip multiprocessors will likely be heterogeneous. It has been suggested by several groups that it may be worthwhile to implement some cores that are specially tuned to execute common code patterns. One such common application that will execute on all future processors is of course the operating system. Many future workloads will spend a large fraction of their execution time within privileged mode, either executing system calls or pure operating system functionality. Vast transistor budgets and relatively low on-chip communication latencies make it feasible to off-load the execution of privileged instruction sequences on to such a custom core. In this paper, we first examine this off-load approach and attempt to understand its benefits. We then try to architect a solution that captures the benefits of off-loading and eliminates its disadvantages. In essence, the benefits of off-loading can be attributed to reduced cache interference, while its disadvantages are the high latency costs for off-load and cache coherence. Our proposed solution employs a special OS cache per core and improves performance by up to 18% for OS-intensive workloads without any significant addition of transistors. We consider several design choices for this OS cache and argue that it is a better use of transistor and power budget than the off-loading approach when both adding to the transistor budget or leaving it unchanged.

1 Introduction

In the era of plentiful transistor budgets, it is expected that processors will accommodate tens to hundreds of processing cores. With processing cores no longer being a luxury, we can consider dedicating some on-chip cores for common applications. The customization of these cores can allow such applications to execute faster and more power-efficiently. There are already several hints that industry is headed towards such heterogeneous chip multiprocessor (HCMP) platforms [5, 8, 9, 11, 16, 18]. One common application that may benefit from customization is the operating system: it executes on every processor and is frequently invoked either by applications or simply to perform system-level book-keeping. The operating system is an especially important target because several studies [1, 6, 14, 18] have shown that the past decade of microarchitectural innovations have done little to boost the performance of OS execution. This is attributed to many factors: OS calls are short, have hard-to-predict branches, have little instruction-level parallelism (ILP), and suffer from cache interference effects. It can also be argued that current high-performance cores are over-provisioned for OS execution (for example, floating-point units, large reorder buffer, large issue width) and are hence highly inefficient in terms of power consumption. Studies [5, 18, 21] have shown that operating system code constitutes a dominant portion of many important workloads such as webservers, databases, and middleware systems. Hence, optimization of OS execution (perhaps with a customized core, henceforth referred to as the *OS core*) has the potential to dramatically impact overall system performance and power.

Some research groups [5, 16, 18] believe in the potential of core customization within multi-cores to improve OS efficiency. In this paper, we put that hypothesis to the test. We first characterize OS behavior and attempt to design a highly optimized off-load mechanism. Based on our observation of cache behavior during the off-load process, we conclude that performance can be optimized with a selective off-load mechanism. We design a predictor to dynamically determine if OS execution should be off-loaded to its own separate core. Thus, our mechanism is a significant advancement of the state-of-the-art. However, there remains room for improvement. Our analysis shows that many cycles are wasted because of costly cache coherence operations between the user core and the OS core. We also observe that modern off-load latencies of 1000+ cycles are an impediment to performance.

To overcome these disadvantages, we explore an alternative path to efficient OS execution. A large fraction of off-loading benefit can be attributed to reduced interference within the caches of the OS and user cores. Off-loading is inefficient because to get this benefit, it forces code migration to a distant core, while also shipping non-trivial amounts of data between these cores. Instead, we propose to bring this extra cache space to the user core itself. We consider several design options while incorporating this extra storage into a

core's cache hierarchy, referred to as the *in-core* approach. We develop dynamic block placement and block look-up policies to control how cache space is allocated between user and OS working sets. We observe that the *in-core* approach out-performs the *off-loading* approach.

While the use of an OS cache per core is a significant departure from the off-loading approach, we believe that it is a better use of a processor's transistor and power budget. We show that a separate OS core cannot simultaneously handle requests from several threads, hence it is wishful to assume that all OS activity in a multi-core can be relegated to a single small core. All known strategies to improve the power-efficiency of a separate OS core can also be employed dynamically within a user core. We therefore conclude that if an argument must be made for off-loading, it must be accompanied by the development of core customization techniques that cannot be dynamically employed within regular user cores. The paper thus sheds new light on the debate over these contrasting approaches, while enhancing the state-of-the-art for both approaches.

The paper is organized as follows. Section 2 summarizes a variety of existing proposals with connections to this work. We describe our experimental methodology in Section 3 and characterize benchmark behavior. Section 4 evaluates the potential of a new mechanism for OS off-loading. We examine how the benefits of a multi-core design can be integrated into a uni-core solution in Section 5. We provide a summary of these two competing approaches in Section 6 and conclude in Section 7.

2 Related Work

In addition to the related work discussed in later sections, we summarize the most related work in three areas: OS interference and impact on user code, improving CMP efficiency, and hardware support for OS execution.

2.1 Operating System Impact on System Throughput

There have been many studies throughout the years on how operating system overhead affects the throughput of user applications (the eventual metric of interest). Chen et al. [7], Thomas et al. [2], and Agarwal et al. [1] have shown that operating system execution generates memory references that negatively impact the performance of traditional memory hierarchies. Redstone et al. [21] and Nellans et al. [18] have shown that there are important classes of applications, namely webservers, databases, and display intensive applications for which the OS can contribute more than half the total instructions executed. Nellans et

al. [18] and Li et al. [14] show OS codes underperform user applications by 3-5x on modern processors and suggest that OS code can be run on less aggressively designed processors to improve energy efficiency.

2.2 Improving CMP Efficiency

Symmetric chip multiprocessors have become commonplace and most application software has not been able to keep pace by developing threaded applications that gracefully scale to multiple cores. To counteract this under-utilization there are a variety of proposals to either scale back the power consumption of underutilized hardware or improve performance by better matching threads to processing cores. Chakraborty et al. [5] propose migrating computation fragments, arbitrary portions of a dynamic instruction stream, on to dedicated cores within a symmetric CMP. They specifically target system calls by migrating system calls from multiple cores onto a single core. Performance increases are seen due to the symbiotic execution of similar instruction streams on a single core. Kumar et al. [11] as well as others [8, 24] propose integrating heterogeneous processing cores instead of symmetric processors in a CMP arrangement. Threads can then be scheduled onto cores which closely match their computation requirements, improving energy efficiency. Mogul et al. [16] extend this idea to the operating system by pairing aggressive cores with energy efficient cores and migrating system call execution onto the energy efficient core. Long duration migrations allow them to power down the the aggressive core, resulting in a net improvement in system efficiency.

2.3 Hardware Support for OS Execution

Li et al. [12–14] have proposed dynamically tuned hardware to reduce energy consumption while the operating system is executing. Their focus on energy efficiency does not overlap with the performance oriented approaches presented in this paper, but many of their optimizations are applicable to our proposed design. This applicability is discussed in a later section. Brown and Tullsen [4] propose a hardware based mechanism for speeding up thread migration across processors in a CMP system. While they do not specifically target the operating system, any reduction in thread migration overhead has a positive effect on operating system offloading in a multi-core solution. Qureshi et al. [19, 20], Jaleel et al. [10], and Suh et al. [23] provide insight into mechanisms by which shared caches can be dynamically tuned via way-partitioning or insertion policies to improve hit rates. The *in-core* cache allocation mechanisms proposed in Section 5 re-discover many of the insights of these works but in a different context. Our mechanism differs from these proposals in that it operates on a non-shared cache within a single thread of execution, rather than on a shared last level cache with multi-programmed workloads.

3 Methodology and Workload Characterization

Simulator and Benchmarks: Except where noted, all experiments in this paper are modeled using Simics 3.0.x simulating a Sunfire 6500 machine and UltraSPARC III series processors. In this paper we examine a broad range of applications including benchmarks from SPECcpu2006, SPECjbb2005, BioBench, Parsec, and an Apache based webserving benchmark serving 500 concurrent requests. When evaluating a core-migration implementation, it is important to evaluate traditional compute-bound applications as well to ensure that the proposed policy, while beneficial on OS-intensive applications, does not substantially reduce performance on other classes of workloads. We used an unmodified Linux operating system, except for the SPECjbb2005 benchmark which was modeled under Solaris 10. For our benchmarks, all measurements were taken over a window of 5 billion instructions. We believe this is at the lower limit of the sampling period necessary to capture representative operating system behavior beyond system calls. We were able to see as much as 20.2% difference in IPC performance for some benchmarks when reducing the sample size to 100 million instructions but increasing our sampling window to 10 billion instructions resulted in a maximum IPC variation of only 1.2%. Before the sampling period, all benchmarks are fast forwarded to a defined region of interest in functional mode, then warmed up for a minimum of 25 million instructions with our microarchitectural structures enabled.

Because of the extended simulation duration, using a cycle accurate simulation environment like GEMS [15], with Opal and Ruby, is not feasible. For this work we use the default in-order processor model provided by Simics with a locally developed memory system based on g-cache that accurately supports cache-to-cache transfers, parallel and sequential look-up policies, and configurable routing between cache levels. Our cache timing models are based on the results from CACTI 6.0 [17] using a target frequency of 3GHz. We implement a main memory latency of 500 cycles in all experiments.

For this work, the only candidates for off-loading are sequences of instructions that execute in privileged mode. *Privileged mode execution* captures both pure operating system functions such as scheduling and kernel threads, as well as system calls executed on behalf of the user code. We make no distinction between these two types of execution, providing us a clear delineation between operating system and user code that is also OS implementation agnostic. We use the term privileged mode execution and operating system execution interchangeably. To determine the OS/user delineation, we examine the PSTATE register in the SPARC architecture that indicates the privilege mode of the processor. We examined our benchmarks under both Solaris and Linux and found that the number of operating system instructions executed in either OS differed by less than +/-10%. Given the low variance between the two operating systems, we do not attempt to quantify the effects of different operating systems in this study.

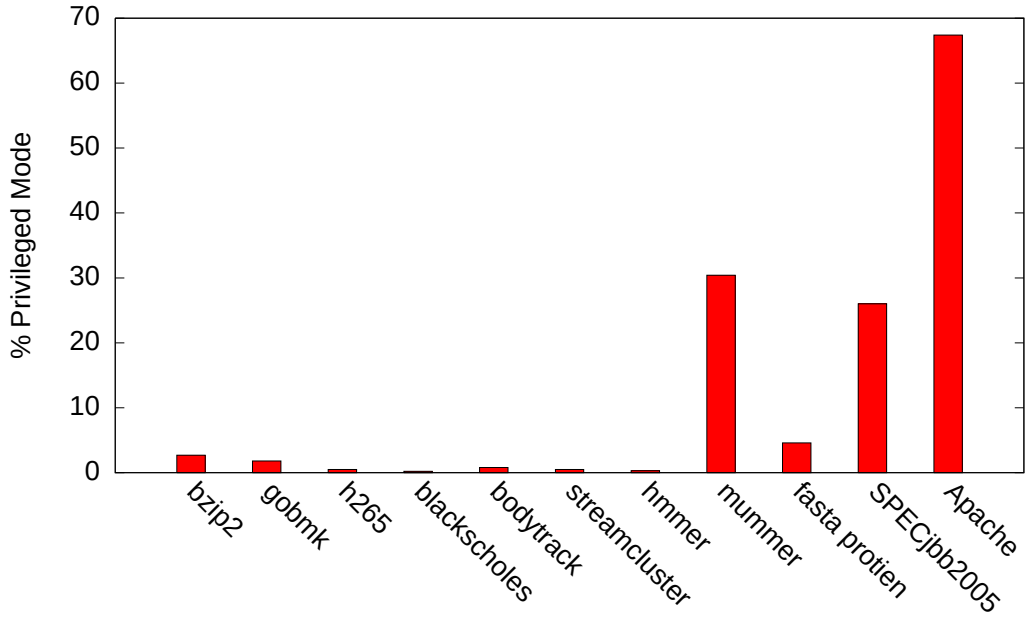

Figure 1: Percentage of Workload Execution Occuring Within Operating System Code (Privileged Mode)

OS Contributions to Execution Time: For separation of operating system and user execution to be beneficial, a significant portion of execution time must occur within the OS. For the sampling provided in Figure 3, all benchmarks are run on a uniprocessor with no other active threads. The total number of privileged instructions varies dramatically, from nearly zero in the compute-intensive benchmarks to as much as 26% in SPECjbb2005 and 67.4% in Apache, our OS-intensive benchmarks. Note that mummer from the Biobench suite seems to be in the same category with 30.4% privileged instructions, however this is an artifact of the SPARC register system which enters privileged mode to rotate the SPARC register windows very frequently. This very short routine accounts for virtually all of the mummer privileged mode instructions. By contrast, register rotate traps make up less than 5% of the total privileged instructions in SPECjbb2005 and Apache (see Figure 2 for a distribution of the length of the privileged instruction runs in terms of number of instructions).

Privileged Instruction Run Lengths: The processor enters privileged mode often for a variety of reasons. As a result, the frequency and duration of these executions can be

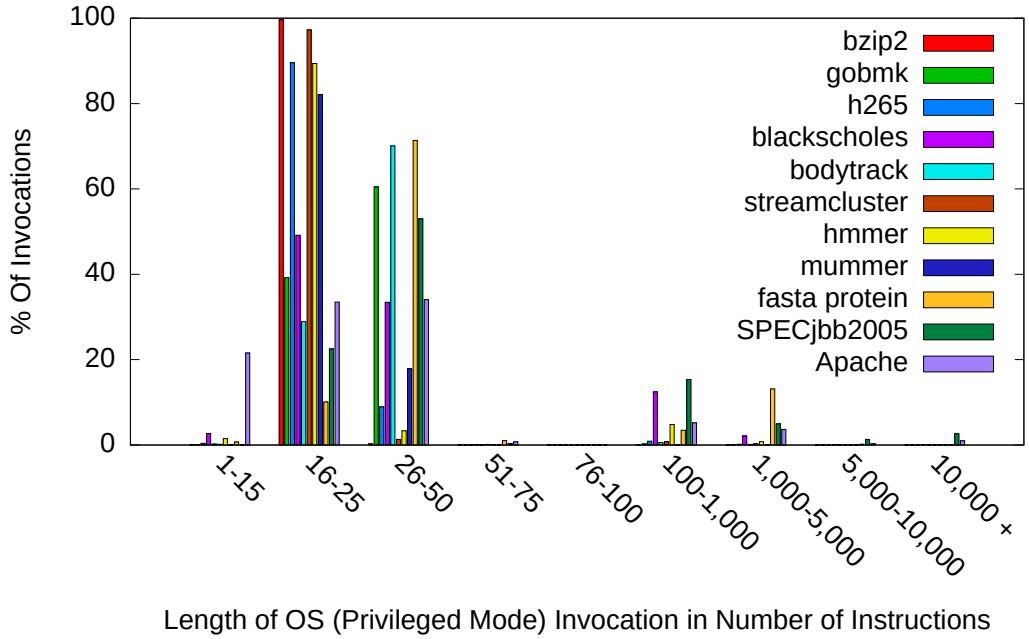

Figure 2: Length of OS system calls and other OS execution during benchmark execution.

wildly different depending on many factors, such as system call use by applications, kernel housekeeping, and device interrupts. Figure 2 shows the duration of privileged instruction runs during a 5 billion instruction window. Both SPECcpu2006 and Parsec have very few significant privilege mode sequences of long duration, which is not surprising given that they are designed to measure CPU throughput and not interact with I/O devices or saturate memory bandwidth. Most benchmarks show by far the largest number of privileged mode invocations last less than 25 instructions. Again, this appears to be unique to the SPARC architecture due to its rotating register file mechanism. Aside from those executions, most remaining privileged instruction sequences are less than 1000 instructions in length, with only SPECjbb2005 and Apache having a non-trivial number of executions over 1,000 instructions long. SPECjbb2005 has one of the largest memory footprints and performs a significant number of system calls through the execution of the JVM for both I/O, thread scheduling, and garbage collection. For the sake of brevity we will provide average results for the nine compute bound benchmarks in the remainder of this paper.

Cache Evictions Caused by OS Execution: The operating system interferes in all levels of the cache hierarchy by causing evictions because of the blocks it brings in. Conversely, those evicted lines of user data can cause a secondary eviction of the OS data if they must be brought back on chip during user execution. Figure 3 shows the causes of OS induced eviction at the L1 and L2 levels. Apache and SPECjbb2005 attribute between 25-30%

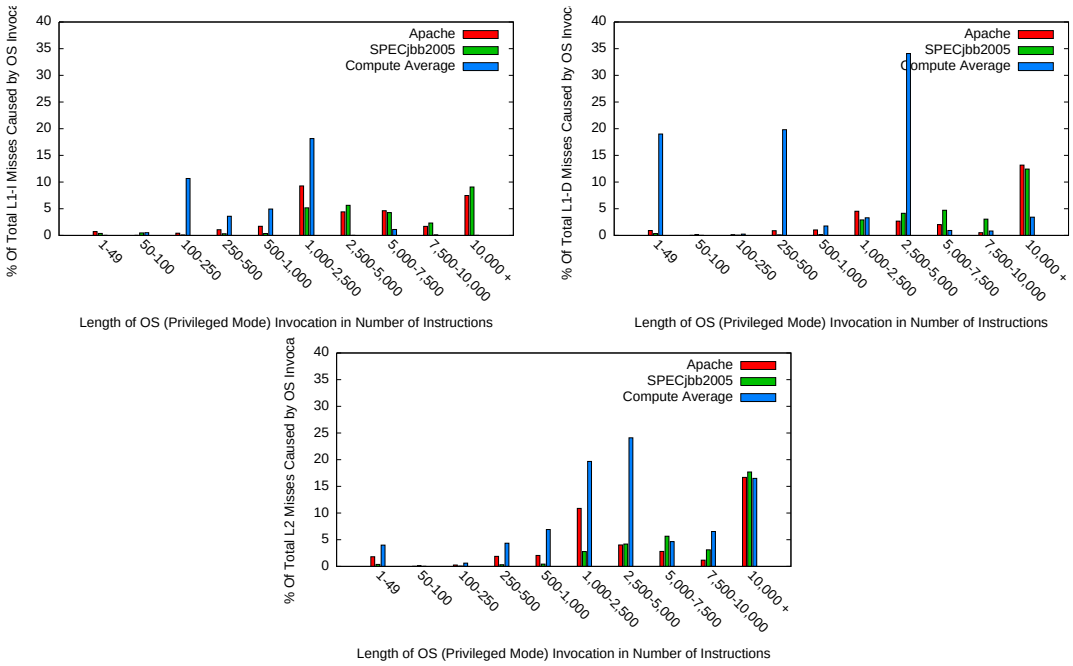

Figure 3: Percentage of total evictions caused by the operating system, broken down by OS execution length.

of total evictions at the L1 level to operating system interference and 35-42% at the L2 level. While the misses caused by the OS in compute-intensive applications is high as a percentage of the total, the absolute number of misses is relatively low because most of these applications are cache-resident.

The above workload characteristics play a strong role in our design decisions for the *OS core* and *OS cache* optimizations. The length of the OS invocation is a key factor in determining whether off-load should happen or not. The removal of cache interference is clearly the crux of both optimizations and the above results show that significant cache interference exists and is related to OS syscall length.

4 Reduced Cache Interference Through Multi-core Offloading

The results in the previous section show that cache interference from the OS is a major problem for OS-intensive workloads. A few groups have argued that off-loading OS syscalls to a customized *OS core* can yield performance and power efficiency [5, 16, 18].

We therefore attempt to design a dynamic off-load policy that is cognizant of syscall length and cache interference effects. We first articulate the overheads for execution migration. Given the likely high cost for such migration, we employ off-loading only if the syscall length exceeds a certain value (referred to as the *off-load trigger*). We then show that off-loading behavior is highly sensitive to the choice of trigger for each program: aggressive off-loading is required to avoid cache interference in some programs, but it also then entails a high off-loading and cache coherence penalty. To design a robust off-load mechanism, we need an estimation of that program's optimal switch trigger and a hardware predictor that estimates the length of an OS system call. We therefore build upon the state-of-the-art in several ways. Unlike the policies of Mogul et al. [16] and Chakraborty et al. [5] that use static off-loading with OS instrumentation, our off-load policies are dynamic and dictated by a hardware predictor. We also pay closer attention to the trade-offs introduced by switch trigger, cache coherence, and migration delays, all of which feed into the design of an optimal policy. Further, note that the goals of the policies of Mogul et al. [16] are focused on power optimizations and are hence subject to different constraints (their results show a reduction in overall performance because of off-loading).

4.1 Overheads of Migration

Thread migration minimally requires interrupting program control flow and writing architected register state to memory on the user processor. The OS core must then be interrupted if it is executing something else, it reads this architected state from memory, and resumes execution. If there is data in cache on the user processor that must be accessed by the OS core, it must be transferred to the OS core, resulting in coherence traffic. Typically, cache values are not aggressively prefetched into the OS core to avoid pollution and wastage – they are fetched on a demand basis, leading to longer latencies per access. On modern hardware, the minimal process migration cost is approximately 5,000-10,000 cycles [16]. Some recent work [4] suggests that hardware support can lower the basic execution migration cost, but there is little that can be done to hide the delays of cache coherence traffic. Since we later show that off-loading is inferior to the in-core approach, our study makes several optimistic assumptions for the off-loading approach to strengthen the confidence in our conclusions. Hence, we also show results where the migration cost is optimistically much lower.

Migration overheads are also impacted by the specific implementation that determines whether to off-load or not. Previous proposals have involved a VMM to trap OS execution and follow a static policy based on off-line profiling [5], or have instrumented the operating system at various entry points to predict the duration it will be exercised [13, 16]. These static software-based mechanisms may require hundreds of cycles, whereas our predictor-

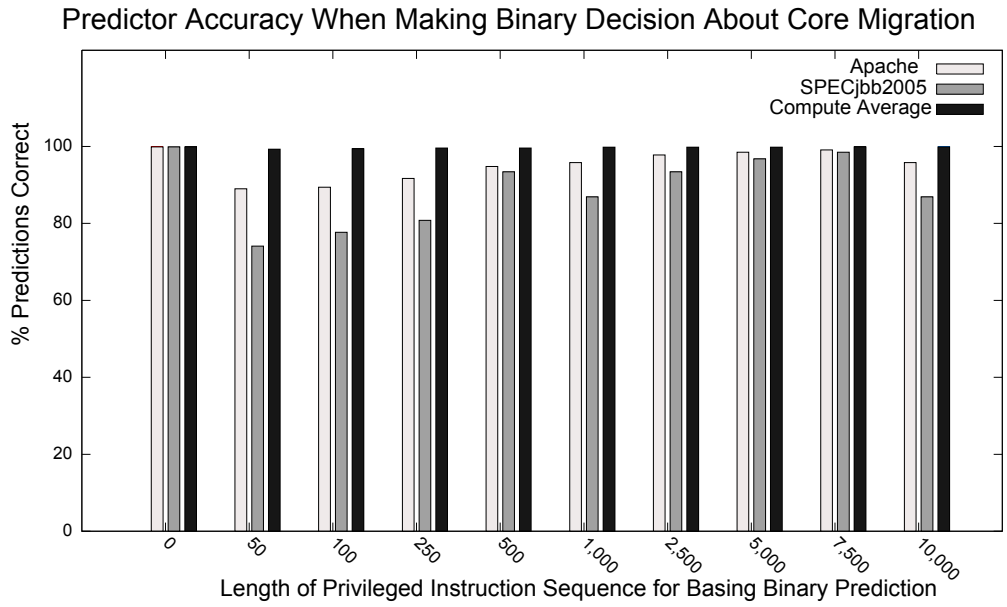

Figure 4: Binary Prediction Hit Rate for Core-Migration Trigger Thresholds

based mechanisms rely on hardware support that can accomplish the decision-making in a single cycle. While this difference may be dwarfed by the long duration of some off-loads, it is a major overhead when considering offloading the most common OS invocations which Figure 2 shows us is very short.

4.2 Hardware Prediction of Privileged Instruction Sequence Length

In order to hide its high opportunity cost, operating system off-load must happen only when executing syscalls that are sufficiently long. Even a single syscall can have dynamic behavior based on its inputs, for example, syscalls performing I/O may have to read a single byte or millions of bytes. Hence, the design of an optimal dynamic off-load mechanism is contingent on our ability to accurately predict the length of a syscall.

We contribute a new hardware predictor of syscall length that simply XOR hashes the values of various architected registers. The following registers were chosen for the SPARC architecture: PSTATE (contains information about privilege state, masked exceptions, FP enable, etc.), g0 and g1 (global registers), and i0 and i1 (input argument registers). The XOR of these registers yields a 64-bit value (referred to as *AState*) that we believe captures information about the type of syscall, input values, and the surrounding environment. The AState value is used to index into a predictor table that keeps track of the syscall length the last time such an AState index was observed. Each entry in the table also maintains

a prediction confidence value, a 2-bit saturating counter per entry that is incremented on a prediction within 5% of the actual, and decremented otherwise. If the confidence value is 0, we found that it is more reliable to make a "global" prediction, *i.e.*, we simply take the average run length of the last three observed syscalls (regardless of their AStates). We observed that this works well because relatively short and similar syscalls tend to be clustered and a global prediction can be better than a low-confidence "local" prediction. For our simulations, we observed that a fully-associative predictor table with 200 entries yields close to optimal performance. We observed that a direct-mapped RAM structure with 1K entries also provides the same accuracy. The storage requirement for our implemented predictor is only 2 KB.

Averaged across all benchmarks, this simple predictor is able to precisely predict the run length of 71.2% of all privileged instruction runs, and predict within $\pm$5% the actual run length an additional 21.1% of the time. Large prediction errors most often occur when the processor is executing in privileged mode, but interrupts have not been disabled. In this case, it is possible that the privileged mode operation is interrupted by one or more additional routines before the original routine is completed. Our predictor does not capture these events well because they are typically caused by external device interrupts which are not part of the processor state at prediction time.

While the hardware predictor provides a discrete prediction of privileged mode instruction run length, the switch trigger must distill this into a binary prediction indicating if the run length exceeds N and core migration should occur. Figure 4 shows the accuracy of binary predictions for various values of N. For example, if core migration should occur only on syscall run lengths greater than 500 instructions, then our predictor makes the correct switching decision 94.8%, 93.4%, and 99.6% of the time for Apache, SPECjbb2005, and the average of all compute benchmarks, respectively. Figure 4 shows us that a switch policy based on our hardware predictor is extremely accurate, indicating that series of privileged mode instructions do indeed have good predictability. While more space-efficient prediction algorithms likely exist, we observe little room for improvement in terms of predictor accuracy. Most mispredictions are caused by unexpected interrupts that would be difficult to provision for. The high-quality predictor developed here serves as an important piece when developing an optimal off-load policy.

4.3 Performance Evaluation of Multi-core Offloading

We next evaluate a predictor-directed off-load mechanism. On every transition to privileged mode, the run-length predictor is looked up and off-loading happens if the run-length is predicted to exceed N (we show results for various values of N). For this experiment, the

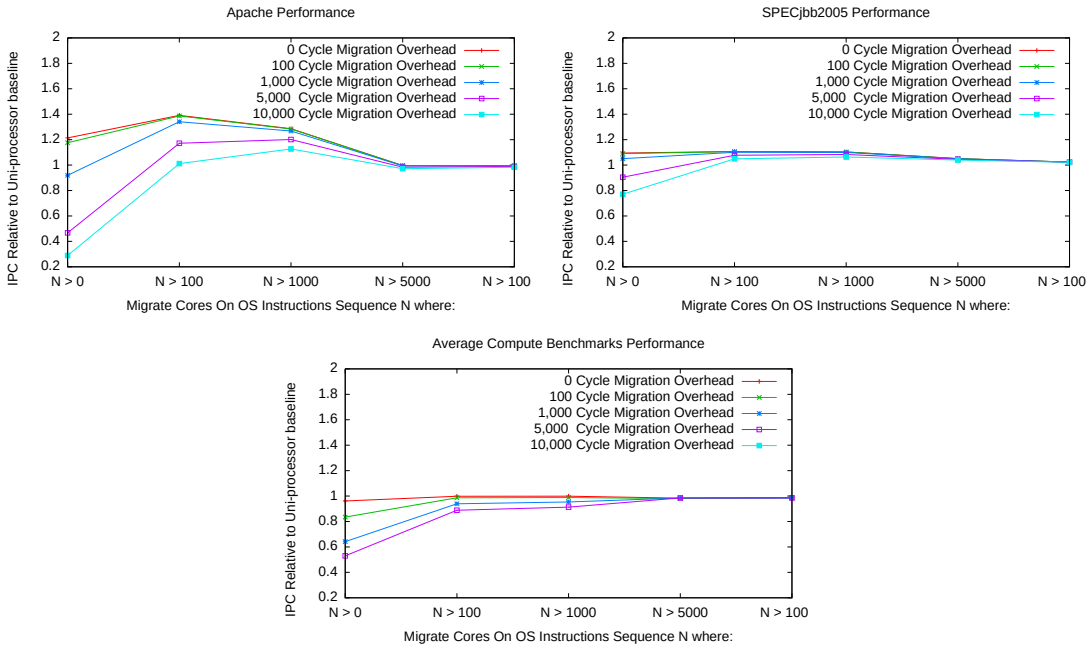

Figure 5: Performance relative to uni-processor baseline when varying the core-switch penalty and the switch trigger policy that performs off-loading only when the syscall length exceeds the given threshold.

	Core Migration Threshold N				
Benchmark	**0**	**100**	**1000**	**5000**	**10,000+**
SPECjbb2005	38.84%	34.48%	33.15%	21.28%	14.79%
Apache	64.7%	45.75%	37.96%	17.83%	17.68%

Table 1: Percentage of Total Execution Time Spent on OS Processor When Using Selective Migration Based on Threshold N

following cache hierarchy was assumed. Cores are symmetric and each core has 1-cycle 32 KB L1 (instruction and data) caches. A miss in the L1 looks up a private L2. This L2 has a capacity of 1 MB and a 5-cycle access time. A miss in the L2 causes a look-up of a directory to detect if the request can be serviced by the L2 of a different core (a coherence miss). If this happens, we assume 5 cycles to look up the remote L2 and an additional 5 cycles for directory look-up and inter-core messaging. The latter 5-cycle penalty is rather optimistic since the directory is typically large and multiple on-chip network messages are required.

Figure 5 shows the IPC performance through off-loading, relative to a baseline that executes the program on a single core. A different graph is shown for Apache, SPECjbb, and compute-intensive programs. Each graph shows the switch trigger threshold on the X-axis

and a different curve for various execution migration delays. For most low migration costs, off-loading is beneficial for a switch trigger as low as $N = 100$ for Apache and SPECjbb. For a high migration cost of at least 5,000 cycles, the optimal switch trigger moves to $N = 1000$ for Apache and SPECjbb. Compute-intensive programs yield almost no benefit with off-loading and are optimized with large N. These programs invoke a number of short syscalls and the overheads of migration do not overcome the benefits of reduced cache interference. Table 1 shows the amount of off-load for the two OS-intensive programs for various values of N. For a migration overhead of 5,000 cycles (achievable on modern systems [16]), off-loading yields an optimal speedup of 1.20 for Apache and 1.08 for SPECjbb. In Apache and SPECjbb, about 4-27% of L2 cache misses are serviced by cache-to-cache transfers. Assuming that remote caches have the same access latency as local caches, *i.e.*, if there is a low-cost coherence mechanism, performance could be improved at most by 3-8%.

4.4 Dynamic Migration Policy Based on Feedback Mechanisms

The second component of our migration mechanism is the estimation of N that yields optimal behavior in terms of say, performance or energy-delay product (EDP). This portion of the mechanism occurs within the operating system at the software level so that it can utilize a variety of feedback information gleaned from hardware performance counters. For this estimation of N, we rely on algorithms developed in prior work to select an optimal hardware configuration [3]. If the hardware system must select one of a few possible hardware configurations at run-time, it is easiest to sample behavior with each of these configurations at the start of every program phase and employ the optimal configuration until the next program phase change is detected.

These algorithms measure various statistics over every time interval (referred to as epochs). Typically, branch mispredict rate, IPC, and cache miss rates are tracked across epochs via performance counters and a significant change in one of these parameters usually signals a phase change. Once a phase change is detected, we execute the application with $N = 100$ for an epoch, then with $N = 500$ for the next epoch, and so on, until all reasonable values of N have been sampled. The value of N with the highest IPC or lowest EDP is then employed until the next phase change is detected. Such a mechanism works poorly if phase changes are frequent. If this is the case, the epoch length can be gradually increased until stable behavior is observed over many epochs. For most of our benchmark programs when looking at epochs larger than 100 million instructions, there were few phase changes and it was fairly straightforward to predict the optimal value of N based on some initial sampling.

5 Reduced Cache Interference Through Intelligent In-Core Caching

Off-loading privileged mode instruction sequences to a separate core yields better performance because of the reduction in cache interference effects. However, we believe that from a performance viewpoint, it is likely more advantageous to employ a single core and augment its cache design than incur the off-load penalty to leverage a remote cache. In this section, we explore *in-core* solutions that add innovations to the cache hierarchy within a core to reduce OS/user cache interference.

The basic idea is simple. Just as the use of separate instruction and data caches is commonplace today, we believe that a separate OS cache per core may be worthwhile. We explored several organizations that adopt an L1 OS cache, but found negligible performance improvements. The small size of L1 caches can not be sub-partitioned without incurring a substantial performance penalty due to increased capacity misses. For the experiments in this section, our uni-processor design implements traditional split L1 instruction and data caches that are shared by both OS and User code.

Our experiments yield different results for the L2 cache. We architect a solution that employs a conventional unified L2 cache for user instructions and data and augment that solution with a special operating system cache. The addition of a last level cache increases power consumption by as little as 0.75 W per MB of cache, whereas the addition of separate core logic, even a simple Alpha EV7-like core, increases power consumption by 7 W [16, 22]. Hence, this is also the more efficient option power-wise, assuming that a single OS core cannot be shared by several user cores in a scalable manner (more on this shortly).

The first design choice we make is to implement mutual exclusion between the contents of the user-L2 cache and the OS-L2 cache. We also considered a solution that allows blocks to be replicated in both caches, but the benefits were nearly non-existent and cache coherence overheads would have to be incurred. It is worth noting that a cache block of instructions is typically accessed in only one of the two modes (privilege or not), while data blocks are often accessed in both modes due to privilege-restricted I/O operations.

Parallel look-up: When an L1 miss is encountered, one of the two L2 cache banks may contain the data. In parallel look-up mode, both cache banks are simultaneously accessed. This increases the access time of each look-up and the power per access. On an L2 miss, the newly fetched block is placed in the OS-L2 or user-L2 based on our *block placement policy* (to be described shortly).

Serial look-up: Alternatively, on an L1 miss, we can predict which bank is likely to contain the data and look it up first. Individual bank look-ups take less time than a parallel bank access. If data is not found in the first L2 bank, the second L2 bank must be sequentially looked up (so as to preserve the mutual exclusion property). Blocks are never swapped between banks. On an L2 miss, the block is initially placed in the bank dictated by our *block placement policy* where the block remains until it is evicted via standard LRU within that bank. Assuming we can accurately predict the bank that contains data for an L2 request, this mode should consume less time and power than the parallel look-up mode.

We model three different *block placement policies*. The first places all data blocks that are fetched by privileged instructions into the OS-L2 bank and everything else into the user-L2 bank (designated as "Route Data Only" in Table 2). The second places all instruction blocks that are fetched by privileged instructions into the OS-L2 bank and everything else in the user-L2 bank (designated as "Route Instructions Only"). The third places all instruction and data blocks fetched by privileged instructions into the OS-L2 bank and everything else in the user-L2 bank (designated as "Route Instructions and Data"). We will subsequently discuss a few more extensions to these policies that dynamically control the capacity allocated to user and OS. We also considered policies based on OS run-length estimation (just as we did for the off-loading experiments), but these did not out-perform the policies above.

For bank prediction, we employ a predictor that mirrors the block placement policy. For example, if the instruction is privileged and our block placement policy is "Route Data Only", we first look for this data in the OS cache assuming that the data must have been fetched by a privileged instruction. Such a bank predictor has perfect prediction for instructions block look-ups. It only incurs mis-predictions when the OS or user attempts to access data that was first brought into the cache by the opposite privilege mode. There are ways to improve upon such bank predictors, for example, by using the instruction PC to index into a table that predicts where this instruction last found its data [25]. Our simple predictor provides an average accuracy of 81% with zero storage overhead, so we did not consider more sophisticated bank predictors in this study.

5.1 Improvements with a Larger Transistor Budget

We consider two sets of experiments. As usual, we assume a baseline core that employs a 1 MB 5-cycle L2 cache. We first show results for an organization that assumes a 1 MB user-L2 combined with a 1 MB OS-L2. While this design clearly has more resources than the baseline, it is similar in spirit to the off-load approach where we attempt to improve performance by increasing the transistor budget. This experiment also allows us to make

Benchmark	Route Data Only	Route Instructions Only	Route Instructions and Data
Parallel Lookup			
ComputeAVG	1.04	.99	1.03
SPECjbb2005	1.35	1.05	1.33
Apache	1.53	1.42	1.49
Sequential Lookup			
ComputeAVG	1.05	1.00	1.03
SPECjbb2005	1.36	1.05	1.32
Apache	1.53	1.45	1.50

Table 2: IPC performance relative to baseline by implementing secondary OS L2 cache

a direct comparison against the off-load results. As is the norm, all L2 caches perform sequential tag and data access. All of our cache delay estimates are obtained with CACTI 6.0. A parallel look-up into both 1 MB banks has an access time of 8 cycles. A sequential look-up with a correct bank prediction takes 6 cycles (this is similar to a 1 MB cache access plus the wire delay to route the request to the correct bank). A sequential look-up with a bank mispredict takes 10 cycles according to CACTI (since we must route the request to the predicted bank, perform a tag look-up in the predicted bank, discover a miss and route the request to the second bank, and then finally perform tag and data look-up over the 1 MB bank).

Table 2 shows results with parallel and sequential look-up for the three bank placement policies. Thanks to the extra cache space, performance is almost always better than the baseline. For the two OS-intensive applications the improvements are in the range of 36% to 53%. This implementation is very similar to the multi-core approach described in Section 4 but we have eliminated 100% of the offloading penalty, thus substantially improving performance. We hypothesized that optimal performance would occur by routing OS instruction and data blocks to the OS cache, but our results show that there is substantial interference between OS data and instruction references in a shared cache. Thus, we observe that optimal performance is seen by separating only the OS data references, allowing instruction blocks to share space with user data blocks. Thanks to the high bank prediction accuracy, we observe that sequential look-up performs equal to or slightly better than parallel look-up in every case.

Note that having the two cache banks co-located opens up the door for creative bank placement policies (for example, with the multi-core off-load solution, it would be highly inefficient to place OS instructions in the user core's cache).

Benchmark	Route Data Only	Route Instructions Only	Route Instructions and Data
Parallel Lookup			
SPECjbb2005	1.17	.91	1.15
Apache	1.02	.93	.99
Sequential Lookup			
SPECjbb2005	1.18	.96	1.16
Apache	1.03	.93	.99

Table 3: IPC performance of a cache hierarchy with as many transistors as the baseline (512 KB OS and 512 KB user)

5.2 Improvements with an Identical Transistor Budget

To make a more fair apples-to-apples comparison against the baseline, we also model the case that has 512 KB banks for the OS and user L2 caches. This effectively partitions an existing L2 design into two equal sub-pieces for OS and user execution. Correspondingly, the parallel look-up now takes 5 cycles, and the sequential look-up takes 4 or 7 cycles depending on whether the bank prediction is correct or not.

The results in Table 3 show us that sequential look-up again outperforms parallel look-up. For SPECjbb2005 we are able to achieve a performance improvement of 19% given the same transistor budget, while Apache improves only a modest 3%. Again data block separation yields the best results; an instruction only block placement policy decreases performance across the board due to underutilization of the OS cache.

For the compute-intensive applications, we observed that the default bank placement policies can yield significant slowdowns compared to the baseline. This is because we are statically allocating half the cache space to user and OS and this is poor allocation for compute-bound applications (and many others) that engage in little OS activity. This under-utilization of the OS cache is much more expensive in these experiments that are increasingly storage-constrained. To remedy this common case, we slightly alter our bank placement policy. If fewer than 25% of all instructions in the last one million instruction epoch were privileged instructions (in other words, there was little OS activity), the bank placement policy alternately uses either bank to place a newly fetched block. This new policy can be easily implemented using only a few hardware counters and minimal logic. Additionally, when using such a round robin allocation, sequential look-up yields no performance improvement because there is no basis for L2 bank predictability. We therefore default back to parallel look-ups. The average performance of our compute bound applications was only 0.4% below the baseline utilizing this alternate block placement policy.

In a similar vein to a dynamic bank placement policy, one could consider dynamically allocating portions of the cache space to user and OS. All of the above experiments statically allocate half the L2 cache space to the user and OS. To allow a dynamic allocation of space, the ways of a unified cache can be dynamically assigned to either OS or user space. While we haven't considered these policies for this paper, we expect that already existing policies such as UCP (utility-based cache partitioning) [10, 20] would help determine what fraction of the 1 MB L2 cache is designated as OS and user.

6 Discussion

Performance Summary: The paper first builds upon the state-of-the-art in off-loading mechanisms by estimating an optimal switch trigger per application and then dynamically predicting if off-loading must happen or not. We make a number of optimistic assumptions for this scheme and show that by adding another symmetric core, performance of Apache and SPECjbb is improved by 20% and 8%, respectively. We then architect an in-core solution: a split L2 cache hierarchy within a core to capture the benefits of off-loading without incurring the latency penalties of execution migration and cache coherence. We explore various bank placement policies, bank prediction, and sequential/parallel look-up policies. If the proposed model is allowed to grow the transistor budget, then by employing many fewer transistors than the off-load case, we observe performance improvements of 55% and 37% for Apache and SPECjbb, clearly better than even the most optimistic off-load improvements. If we assume that total cache capacity in the new solution is not allowed to exceed the cache capacity in the baseline, the in-core solution yields improvements of 3% and 18%, compared to the -15% and -19% improvements (degradations!) for the off-load solution. Note that in all these comparisons, the off-loading approach has a larger transistor budget than the in-core approach because the former implements a second core.

Power Comparison: It has been argued that power can be saved by off-loading OS execution to a low-power core. When off-loading happens, the user core moves to a low-power mode and similarly, the OS core moves to a low-power mode when it is idle. To allow these transitions to happen in tens to hundreds of cycles, only frequency changes can be considered (voltage transitions take several thousand cycles in most implementations). As a result, these low-power modes do little to reduce leakage, which can be as much as 25% of processor power [22]. If the OS and user cores are symmetric, this means that the combination dissipates at least 1.25 times the power of the single-core baseline at any time. As noted earlier, performance speedups are less than 1.25x. Of course, with an asymmetric low-power OS core, the off-loading approach can yield improvements in EDP. However,

note that a similar power-down strategy can also be adopted for the uni-core approach. If we are willing to execute privileged OS instructions at a much lower performance level, the single core can scale back its frequency by as large a factor as desired in the matter of cycles (the Intel Montecito can do dynamic frequency scaling in less than a handful of cycles) every time a privileged syscall is made. Thus, while both approaches can incur equal power penalties from the extra cache, the off-load approach suffers because one of its cores is always leaking power as it idles. It is certainly possible that with the right microarchitectural changes, the OS core can be designed to be so power-efficient that it can deliver better EDP than the in-core solution with frequency scaling, but this is likely at unacceptable performance overheads.

Scalability: The other argument in favor of the off-load approach is that it may be more efficient (in terms of performance, transistors, and power) to implement a single OS core that is shared by all cores on chip. This allows multiple OS syscalls (invoked by different threads) to symbiotically maintain a shared working set in the OS core's cache, thus possibly boosting performance. Power overheads are also minimized since only one additional core (and its leakage) is being introduced. However, we observed that the OS core has very poor scalability when handling OS-intensive programs. As shown in Table 1, the OS core is heavily utilized. This means that a syscall often has to stall because the OS core is busy executing a syscall invoked by another core. These stall times were frequently of the order of 1000 cycles when executing two SPECjbb threads and exploded to 25,000 cycles for four threads (the OS core is basically saturated by requests from as few as four cores). Hence, the off-loading approach has very limited scalability for OS-intensive workloads and negligible improvements for non-OS-intensive workloads.

Thus, across all metrics, we believe that the in-core approach is superior to the off-load approach.

7 Conclusions

This paper considers two competing approaches to handle the execution of privileged OS instructions. Our focus is on performance, and primarily the boost afforded by reducing OS-user interference within the cache. We consider several novel innovations to each approach. For the off-loading approach, we introduce an adaptive off-load policy based on behavior profiling and syscall run-length prediction. While this yields good benefits, we argue that there is room for improvement. We propose a novel competing in-core approach that hasn't been previously considered. We introduce a cache within a core to cache a

subset of OS references and consider several design options for it, including various block placement policies, bank predictors, and sequential/parallel look-ups. This option yields significant improvements: up to 53% if one is willing to increase the transistor budget (similar to the off-loading approach which adds a separate core), and up to 18%, while preserving the same transistor budget as the baseline. We also show that this solution is superior to the off-load approach in terms of performance, transistor budgets, and power.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 1988.

[2] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The Interaction of Architecture and Operating System Design. In *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Santa Clara, CA (USA), 1991.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Managing the Communication-Parallelism Trade-Off in Future Clustered Processors. In *Proceedings of ISCA-30*, pages 275–286, June 2003.

[4] J. A. Brown and D. M. Tullsen. The Shared-Thread Multiprocessor. In *Proc. ICS*, pages 73–82, 2008.

[5] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-Fly. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 283–292, New York, NY, USA, 2006. ACM.

[6] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Symposium on Operating Systems Principles*, pages 120–133, 1993.

[7] N. Gloy, C. Young, J. B. Chen, and M. D. Smith. An Analysis of Dynamic Branch Prediction Schemes on System Workloads. In *Proc. 23rd Annual Intl. Symp. on Computer Architecture*, pages 12–21, 1996.

[8] R. Grant and A. Afsahi. Power-performance Efficiency of Asymmetric Multiprocessors for Multi-threaded Scientific Applications. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.

[9] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, 2005.

[10] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, J. Simon Steely, and J. Emer. Adaptive Insertion Policies For Managing Shared Caches. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 208–219, New York, NY, USA, 2008. ACM.

[11] R. Kumar, K. I.Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *MICRO, Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[12] T. Li and L. John. OS-aware Tuning: Improving Instruction Cache Energy Efficiency on System Workloads. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, pages 10–22, 2006.

[13] T. Li, L. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio. Understanding and Improving Operating System Effects in Control Flow Prediction. *Microprocessors and Microsystems*, June 2006.

[14] T. Li and L. K. John. Operating System Power Minimization through Run-time Processor Resource Adaptation. *IEEE Microprocessors and Microsystems*, 30:189–198, June 2006.

[15] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[16] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *Micro, IEEE*, 28(3):26–41, May-June 2008.

[17] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO-40)*, December 2007.

[18] D. Nellans, R. Balasubramonian, and E. Brunvand. A Case for Increased Operating System Support in Chip Multi-processors. In *IBM Annual Thomas J. Watson $P=ac^2$ Conference*, 2005.

[19] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 381–391, New York, NY, USA, 2007. ACM.

[20] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[21] J. Redstone, S. J. Eggers, and H. M. Levy. An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In *Architectural Support for Programming Languages and Operating Systems*, pages 245–256, 2000.

[22] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, B. Cherkauer, J. Stinson, J. Benoit, R. Varada, J. Leung, R. Lim, and S. Vora. A 65-nm Dual-Core Multithreaded Xeon Processor With 16-MB L3 Cache. *IEEE Journal of Solid State Circuits*, 42(1):17–25, January 2007.

[23] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *J. Supercomput.*, 28(1):7–26, 2004.

[24] B. Wun and P. Crowley. Network I/O Acceleration in Heterogeneous Multicore Processors. *14th IEEE Symposium on High-Performance Interconnects (HOTI'06)*, 0:9–14, 2006.

[25] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proceedings of ISCA-26*, pages 42–53, May 1999.