Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionality

David Nellans, Kshitij Sudan, Erik Brunvand, Rajeev Balasubramonian

School of Computing, University of Utah
{*dnellans, kshitij, elb, rajeev*}*@cs.utah.edu*

Abstract. Modern and future server-class processors will incorporate many cores. Some studies have suggested that it may be worthwhile to dedicate some of the many cores for specific tasks such as operating system execution. OS off-loading has two main benefits: improved performance due to better cache utilization and improved power efficiency due to smarter use of heterogeneous cores. However, OS off-loading is a complex process that involves balancing the overheads of off-loading against the potential benefit, which is unknown while making the off-loading decision. In prior work, OS off-loading has been implemented by first profiling system call behavior and then manually instrumenting some OS routines (out of hundreds) to support off-loading. We propose a hardware-based mechanism to help automate the off-load decision-making process, and provide high quality dynamic decisions via performance feedback. Our mechanism dynamically estimates the off-load requirements of the application and relies on a run-length predictor for the upcoming OS system call invocation. The resulting hardware based off-loading policy yields a throughput improvement of up to 18% over a baseline without off-loading, 13% over a static software based policy, and 23% over a dynamic software based policy.

1 Introduction

In the era of plentiful transistor budgets, it is expected that processors will accommodate tens to hundreds of processing cores. Given the abundance of cores, it may be beneficial to allocate some chip area for special-purpose cores that are customized to execute common code patterns. One such common code is the operating system (OS). Some prior studies [10, 17, 19] have advocated that selected OS system calls be off-loaded to a specialized OS core. This can yield performance improvements because (i) user threads need not compete with the OS for cache/CPU/branch predictor resources, and (ii) OS invocations from different threads interact constructively at the shared OS core to yield better cache and branch predictor hit rates. Further, in a heterogeneous chip multiprocessor, the OS core could be customized for energy-efficient operation because several modern features (such as deep speculation) have been shown to not benefit OS execution [17, 19].

Hardware customization for OS execution has high potential because the OS can constitute a dominant portion of many important workloads such as

webservers, databases, and middleware systems [10, 19, 21]. These workloads are also expected to be dominant in future datacenters and cloud computing infrastructure, and it is evident that such computing platforms account for a large fraction of modern-day energy use [23]. In these platforms, many different virtual machines (VMs) and tasks will likely be consolidated on simpler, many-core processors [6, 23]. Not only will the applications often be similar to existing applications that have heavy OS interaction, the use of VMs and the need for resource allocation among VMs will inevitably require more operating system (or privileged) execution.

The OS off-load approach has only recently come under scrutiny and more work is required to fully exploit its potential, including a good understanding of server/OS workloads in datacenters, evaluating the impact of co-scheduled VMs, technologies for efficient process migration, technologies for efficient cache placement and cache-to-cache transfers, etc. The work by Brown and Tullsen [9], for example, attempts to design a low-latency process migration mechanism that is an important technology for OS off-load. Similarly, in this paper we assume that OS off-load is a promising approach and we attempt to resolve another component of OS off-load that may be essential for its eventual success, viz, the decision-making process that determines which operations should be off-loaded. While OS off-load has a significant energy advantage [17], this paper primarily focuses on the performance aspect. We expect that our proposed design will also be useful in an off-loading implementation that optimizes for energy efficiency.

Past work has demonstrated the potential of OS off-load within multi-cores for higher performance [10] and energy efficiency [17]. In these works, the fundamental mechanism for OS isolation, *off-loading*, remains the same. Off-loading implementations have been proposed that range from using the OS' internal process migration methods [17], to layering a lightweight virtual machine under the OS to transparently migrate processes [10]. In these studies, the decision process of which OS sequences to off-load has been made in software, utilizing either static offline profiling or developer intuition. This process is both cumbersome and inaccurate. Firstly, there are many hundreds of system calls, as seen in Table 1, and it will be extremely time-consuming to manually select and instrument candidate system calls for each possible OS/ hardware configuration. Secondly, OS utilization varies greatly across applications and off-loading decisions based on profiled averages will be highly sub-optimal for many applications.

This paper attempts to address exactly this problem. We only focus on performance optimization with OS off-loading on a homogeneous CMP. We contribute to the existing body of work by proposing a novel hardware mechanism that can be employed on nearly any OS/hardware combination. At run-time, we dynamically estimate the level of off-loading that will be most beneficial for the application. Off-loading involves non-trivial interactions: reduced cache/branch predictor interference for the user thread, more cache coherence misses for the OS and user, overhead for off-load, queuing delay at the OS core, etc. We must therefore first estimate how aggressive the off-loading mechanism needs to be and then estimate if each OS invocation should be off-loaded. At the heart of our

proposed scheme is a predictor that accurately estimates the length of upcoming OS sequences. We show that such a hardware-based scheme enables significant benefits from off-loading, out-performing static software instrumentation as well as a similar dynamic instrumentation in software. It also greatly simplifies the task of the OS developer, while incurring only a minor hardware storage overhead of about 2 KB.

Table 1. Number of distinct system calls in various operating systems.

Benchmark	# Syscalls	Benchmark	# Syscalls
Linux 2.6.30	344	Linux 2.2	190
Linux 2.6.16	310	Linux 1.0	143
Linux 2.4.29	259	Linux 0.01	67
FreeBSD Current	513	Windows Vista	360
FreeBSD 5.3	444	Windows XP	288
FreeBSD 2.2	254	Windows 2000	247
OpenSolaris	255	Windows NT	211

A related alternative to off-loading is the dynamic adaptation of a single processor's resources when executing OS sequences [15, 20]. While the dynamic adaptation schemes do not rely on off-loading, the decision making process on when to adapt resources is very similar to that of making off-loading decisions. We therefore expect the proposed prediction mechanisms to be useful for other forms of OS optimization as well, although this is not evaluated in the paper.

2 Background and Motivation

Historical Perspective. Off-loading execution from a traditional general purpose microprocessor is not a new idea. For instance, in the 1980's floating point hardware often existed as a co-processor that could be plugged into an additional socket on many motherboards. This essentially amounted to off-loading floating-point execution. Until recently, the memory controller for the DRAM sub-system was implemented in the north-bridge, but recent CPUs now have an integrated memory controller. Network packet processing can be done within the CPU or it can be delegated to a dedicated Ethernet controller. Graphics rendering can be done entirely within software on the CPU, or it can be sent to a dedicated video card that can render polygons much more efficiently due to a (vastly) different microarchitecture. Given the abundance in transistors today, off-loading within a multi-core can be attempted as a design optimization for common code patterns such as the operating system.

Benchmarks. For this work we examine a broad variety of workloads to examine the effect that OS interference has on cache performance. We look at a subset of benchmarks from the PARSEC [8], BioBench [3], and SPEC-CPU-2006 [12] suites as representative of HPC compute bound applications. Apache 2.2.6 serving a variety of static webpages selected at random by a serverside CGI script, SPECjbb2005 (a middleware-type system), and Derby (a database

workload from the SPECjvm2008 suite) comprise our server oriented workloads. Our server benchmarks map two threads per core except Apache which self tunes thread counts to optimize throughput. This 2:1 mapping allows workloads that might stall on I/O operations to continue making progress, if possible.

All benchmarks were warmed up for 25 million instructions prior to being run to completion within the region of interest, using throughput as the performance metric. For single threaded applications, throughput is equivalent to IPC. In many experiments, the group of compute bound applications displays extremely similar behavior. For the sake of brevity, we represent these applications as a single group in our graphs, and note any outlier behavior.

Off-loading Decisions and Instrumentation Cost.

We view the work by Chakraborty et al. [10] and Mogul et al. [17] as the state-of-the-art in OS off-loading. In both of these works, off-line profiling and developer intuition is used as a guide to identify what are typically long-running system calls. These system calls are then manually instrumented so that their invocation results in a process migration from the application's core to a dedicated OS core. The reverse migration is performed when the system call completes.

Previous proposals [10] have involved a VMM to trap OS execution and follow a static off-loading policy based on off-line profiling. Manual instrumentation of the code can allow the decision-making process to be smarter as the length of that specific system call invocation can be estimated based on input arguments [14, 17]. For example, the duration of the *read* system call is a function of the number of bytes to be fetched from a file descriptor offset and thus can vary across invocations. Even such instrumentation has several short-comings. In some cases, the read syscall may return prematurely if end-of-file is encountered (for other syscalls too, the input arguments may not always be good predictors of run length). In yet other cases, the system call may be preempted by an additional long OS sequence initiated by a external device interrupt. The above effects cannot be accurately captured by instrumentation alone. Therefore, even sophisticated instrumentation can often be inaccurate. A history based predictor of OS run-length has the potential to overcome these short-comings.

We frequently see one or both of the following patterns: (a) an application that invokes many short OS routines, (b) an application that invokes few, but long running, routines. Depending on the application, reduced cache interference may be observed by off-loading one or both classes of OS routines. As we show later in the Section 5, contrary to what intuition might indicate, it is often beneficial to off-load short system calls, even those with shorter duration than the off-loading migration overhead. When short system calls are also considered as candidates for off-loading, the overhead of instrumentation greatly increases. The latency cost of instrumentation code can range from just tens of cycles in basic implementations to hundreds of cycles in complex implementations. This overhead is incurred even when instrumentation concludes that a specific OS invocation should not be off-loaded. As an example, we instrumented the simple *getpid* syscall implementation in OpenSolaris. We found that adding a single off-loading branch that invokes off-loading functionality based on a static

threshold, increases the assembly instruction count from 17 to 33 for this trivial instrumentation. Examining multiple register values, or accessing internal data structures can easily bloat this overhead to hundreds of cycles which quickly penalizes performance. Figure 1 shows the significant performance impact of instrumenting all OS entry points for the server and compute bound workloads.

The above arguments highlight that profile-based manual instrumentation is not only burdensome, it also significantly limits the potential of the off-loading approach. The upcoming sections show how hardware-based single-cycle decision making can out-perform a couple of reasonable static instrumentation policies.

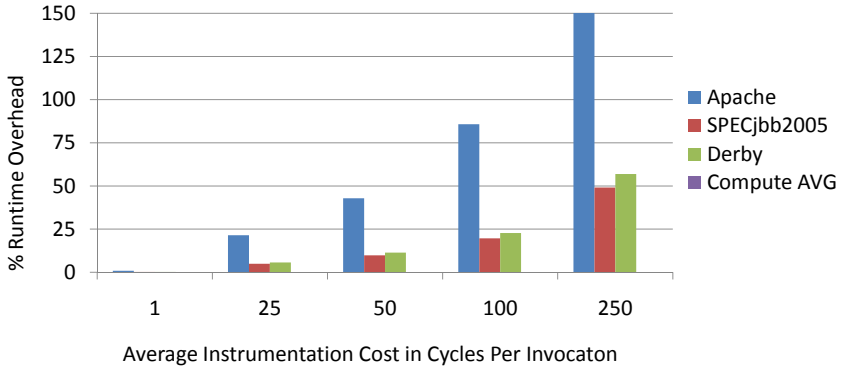

Fig. 1. Runtime overhead of dynamic software instrumentation for all possible OS off-loading points

Migration Implementations. The latency and operations required for off-loading are significantly impacted by the migration implementation, which depends on both hardware and software capabilities. There are other alternatives to process migration, however, such as remote procedure calls, and message passing interfaces within the operating system. These alternate designs have the potential to lower inter-core communication cost substantially and are an interesting design point though we do not consider them in this study. For some research operating systems [7, 13], the notion of off-loading is not even valid, as sub-systems are already pinned to specific processors within a CMP system. However, no mainstream OS has chosen to implement such features yet and by far, the majority of server applications are run on traditional Unix-based systems. In this study, we attempt to be agnostic to the mechanism by which off-loading is performed and show results for a variety of possible off-loading overheads.

While our study evaluates several design points for off-loading overheads, we focus most of our conclusions on the conservative and aggressive design points. The conservative scheme is based on the thread migration time of approximately 5,000 cycles for an unmodified Linux 2.6.18 kernel. Proposals exist that could improve this delay to below just below 3,000 cycles on our target machine [22].

Reliable thread migration entails interrupting program control flow on the user processor and writing architected register state to memory. The OS core must then be interrupted, save its own state if it was executing something else, read the architected state of the user core from memory, and resume 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 (automatically handled by the coherence mechanism). The aggressive scheme is based on the technique proposed by Brown and Tullsen [9] and is assumed to incur a 100 cycle migration latency. They advocate hardware support for book-keeping and thread scheduling (normally done in software by an OS or virtual machine). This comes at the expense of an additional state machine to compute and maintain these records.

3 Hardware-based Decision-Making

Instead of a software instrumentation process based on profiled analysis, we propose a hardware-based mechanism that simplifies the task of the OS developer and makes high quality decisions about the benefits of off-loading OS execution. Off-loading leads to several complex interactions. First, it isolates the working sets of the application and the OS by placing most of the application's data in its own cache and most of the OS' data in the OS core's cache. This reduces cache and branch predictor interference at the user core. It also increases the likelihood that an OS system call will find its data in cache because a similar function was recently executed (not necessarily by the invoking application). Second, the number of coherence misses may increase because the OS system call may have to access application data (that resides in the application core's cache) and conversely, the application may have to access data recently fetched by the OS syscall (that resides in the OS core's cache). These interactions are a strong function of system call length and frequency. Performance is also impacted by the overheads for migration and because of queuing delays at the OS core (since a single OS core could handle requests from multiple application cores). Many of the above effects are not easily captured with performance counters, and even if they were, it may be difficult to construct models that relate these parameters to an eventual decision regarding the benefits of off-loading the system call being currently invoked. We therefore adopt a simplified but effective approach that is based on the following two sub-components.

We adopt the simple strategy that a system call will be off-loaded if it is expected to last longer than a specified threshold, N cycles. The first sub-component employs a hardware predictor to estimate the length of a system call. At any point, we also need to determine an optimal value for N. This may vary depending on how closely the application and OS interact and on whether coherence effects dominate over other cache interference effects. The second sub-component of the proposed scheme determines the optimal value of N by sampling various candidate values.

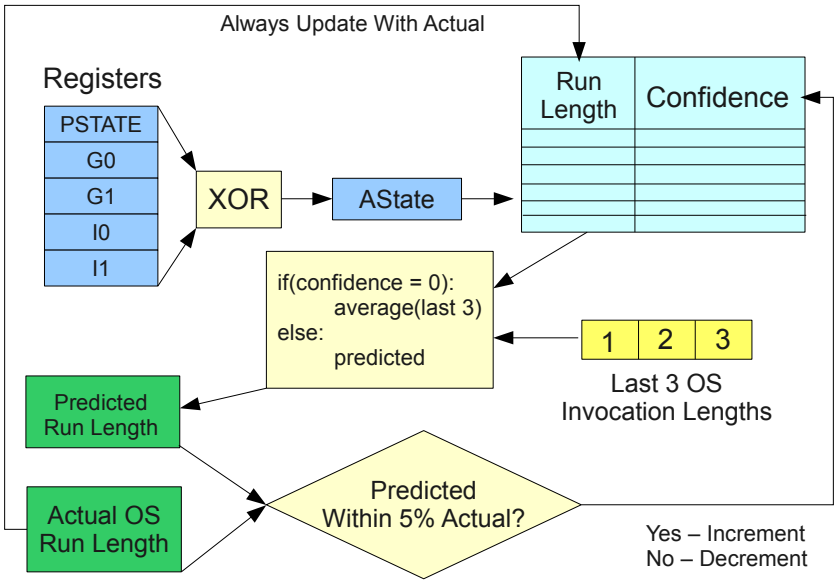

Fig. 2. OS Run-Length Predictor With Configurable Threshold

3.1 Hardware Prediction of OS Syscall Length

We believe that system call run-length is the best indicator of whether off-loading will be beneficial. This is simply because the overhead of migration is amortized better if the system call is longer. The length of the syscall is often a function of the input arguments and processor architected state. We therefore propose a new hardware predictor of OS invocation length that XOR hashes the values of various architected registers. After evaluating many register combinations, 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 (that we refer to as *AState*) that encodes pertinent information about the type of OS invocation, input values, and the execution environment. Every time there is a switch to privileged execution mode, the AState value is used to index into a predictor table that keeps track of the invocation length the last time such an AState index was observed, as shown in Figure 2.

Each entry in the table also maintains a prediction confidence value, a 2-bit saturating counter that is incremented on a prediction within ±5% of the actual, and decremented otherwise. If the confidence value is 0, we find that it is more reliable to make a "global" prediction, *i.e.*, we simply take the average run length of the last three observed invocations (regardless of their AStates). This works well because we observe that OS invocation lengths tend to be clustered and a global prediction can be better than a low-confidence "local" prediction.

For our workloads, we observed that a fully-associative predictor table with 200 entries yields close to optimal (infinite history) performance and requires only 2 KB storage space. The 200-entry table is organized as a CAM with the 64-bit AState value and prediction stored per entry. A direct-mapped RAM structure with 1500 entries also provides similar accuracy and has a storage requirement of 3.3 KB. This table is tag-less and the least significant bits of the AState are used as the index.

Averaged across all benchmarks, this simple predictor is able to precisely predict the run length of 73.6% of all privileged instruction invocations, and predict within ±5% the actual run length an additional 24.8% 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 for privileged mode operation to be 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 devices which are not part of the processor state at prediction time. These prediction inaccuracies are part of non-deterministic execution and can not be foreseen by software or other run length prediction implementations. Fortunately, these interrupts typically extend the duration of OS invocations, almost never decreasing it. As a result, our mispredictions tend to underestimate OS run-lengths, resulting in some OS off-loading possibly not occurring, based on a threshold decision.

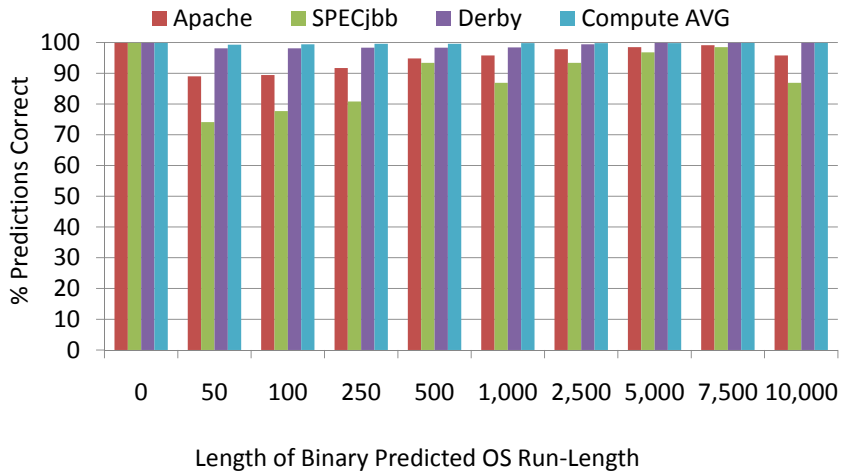

Fig. 3. Binary Prediction Hit Rate for Core-Migration Trigger Thresholds

While the hardware predictor provides a discrete prediction of OS run-length, the off-load decision must distill this into a binary prediction indicating if the run length exceeds N instructions and if core migration should occur. Figure 3 shows the accuracy of binary predictions for various values of N. For example, if off-loading should occur only on OS invocation run lengths greater than 500

instructions, then our predictor makes the correct off-loading decision 94.8%, 93.4%, 96.8%, and 99.6% of the time for Apache, SPECjbb2005, Derby and the average of all compute benchmarks, respectively. While more space-efficient prediction algorithms possibly exist, we observe little room for improvement in terms of predictor accuracy.

3.2 Dynamic Estimation of N

The second component of a hardware assisted off-loading policy 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. Execution of this feedback occurs on a coarse granularity however, typically every 25-100 million cycles. As a result, the overhead is minimal compared to software instrumentation of system calls which can be invoked as often as every few thousand cycles in OS intensive applications.

For this estimation of N, we rely on algorithms described in past work to select an optimal hardware configuration [5]. If the hardware system must select one of a few possible N thresholds 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. The mechanism is epoch-based, *i.e.*, statistics are monitored every *epoch* (an interval of a fixed number of cycles).

For our implementation, where performance is our metric of interest, we use the L2 cache hit rate of both the OS and user processors, averaged together, as our performance feedback metric. Our initial sampling starts with an epoch of 25 million instructions, and an off-loading threshold of $N = 1,000$ if the application is executing more than 10% of its instructions in privileged mode, otherwise the threshold is set to $N = 10,000$. We also sample values for two alternate N, above and below the initial N. If either of these N results in an average L2 hit-rate that is 1% better than our initial N, we set this new value as our threshold. Having chosen an initial threshold value, we then allow the program to run uninterrupted for 100 M instructions. We then again perform a 25 M instruction sampling of two alternate values of N. If our threshold appears to still be optimal we then double the execution length (to 200 M) instructions before sampling again to help reduce sampling overhead. If at any point our current N is found to be non-optimal, the execution duration is reduced back to 100 M instructions.

Such a mechanism can work 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. Such a strategy was not invoked for most of our benchmark programs as few phase changes were encountered for epochs larger than 100 million instructions. For our experiments, we use very coarse-grained values of N (as later reported in Figure 4). Increasing the resolution at which

N can vary will increase the performance of the system, but it comes at the expense of increased sampling overhead.

4 Experimental Methodology

To examine the design space of OS off-loading we use cycle accurate, execution driven simulation to model full OS execution during workload execution. Our simulation infrastructure is based on Simics 3.0 [16] and we model in-order UltraSPARC cores. By modeling in-order cores, we are able to simulate large executions in a reasonable amount of time, thus capturing representative OS behavior. It also appears that many OS-intensive server workloads are best handled by in-order cores with multi-threading [21] (for example, the Sun Niagara and Rock designs and the recent Intel Atom design).

On the SPARC platform, the PSTATE register [1] holds the current state of the processor and contains information (in bit fields) such as floating-point enable, execution mode (user or privilege), memory model, interrupt enable, etc. Our proposed techniques use the execution mode bit in this register to determine which code sequences are executing within the boundaries of the OS. Based on this definition, system calls which occur in privileged mode but within the user address space are captured as part of our OS behavior in addition to functionality that occurs within the kernel address space. Thus, compared to prior work, we are considering a broader spectrum of OS behavior as a candidate for off-loading. Previous work examined only system calls, or a subset of them; we show that optimal performance can be obtained by off-loading OS sequences that are much shorter than intuition might indicate. Therefore, a general-purpose solution for capturing all OS execution is required.

The SPARC ISA has several unique features which cause many short duration (<25 instructions) OS invocations. These invocations are exclusively due to the fill and spill operations of the rotating register file the SPARC ISA implements when the register file becomes overloaded. Other architectures, like x86, perform stack push and pop operations in user space. We analyzed our results both including and excluding these invocations for SPARC ISA, and have chosen to omit these invocations from our graphs where they skew results substantially from what would be seen on an alternative architecture.

Table 2. Simulator Parameters.

CPU Parameters		Memory System Parameters	
ISA	UltraSPARC III ISA	L1 I-cache	32 KB/2-way, 1-cycle
Core Frequency	3.5 GHz @ 32nm	L1 D-cache	32 KB/2-way, 1-cycle
Processor Pipeline	In-Order	L2 Cache	1 MB/16-way, dual banked, 12-cycle
TLB	128 Entry Set Assoc.	L1-L2 Line Size	64 Bytes
Coherence Protocol	Dir. Based MESI	Main Memory	350 Cycle Uniform Latency

For all our simulations, Table 2 shows the baseline memory system parameters. Our timing parameters were obtained from CACTI 6.0 [18] targeting a frequency of 3.5 GHz. A memory latency of 350 cycles is used in all experiments (based on real machine timings from Brown and Tullsen [9]). In the case of off-loading, we simulate two such cores with private L2s which are kept coherent via a directory based protocol and a simple point-to-point interconnect fabric (while this is overkill for a 2-core system, we expect that the simulated model is part of a larger multi-core system). Our system models directory lookup, cache-to-cache transfers, and coherence invalidation overheads independently. We parameterize the *migration implementation* so that we can examine the effects of varied latency implementations on the off-loading decision policy.

5 Results

5.1 Impact of Design Parameters

We next evaluate the off-loading performance achievable with our predictor-directed decision making policy. On every transition to privileged mode, the run-length predictor is looked up and off-loading occurs if the run-length is predicted to exceed N (we show results for various static values of N). Figure 4 shows the IPC performance through off-loading, relative to a baseline that executes the program on a single core. All results use our hardware based predictor and have just a single cycle instrumentation cost. A different graph is shown for Apache, SPECjbb2005, Derby, and compute-intensive programs. Each graph shows a static off-loading threshold N on the X-axis and a different curve for various one-way off-loading latencies. Evaluating multiple off-loading latencies is important because they are highly implementation dependent and can range from 5,000 cycles in current operating system implementations, down to just a few hundred cycles in some recent research proposals [22]. Figure 4 helps identify 3 major trends about OS off-loading.

Off-loading latency is the dominant factor in realizing performance gains. Performance is clearly maximized with the lowest off-loading overhead possible. If the core migration implementation is not efficient, it is possible that off-loading may never be beneficial (see SPECjbb).

For any given off-loading latency, choosing the appropriate switch trigger threshold N is critical. When finding the optimal off-loading threshold N, two factors come into play, OS/User cache interference and cache coherence overheads. When the OS and user application execute on a single core, as has traditionally occurred, the working set of each is forced to compete for space in the cache. Off-loading has the potential to improve performance because we are effectively removing the accesses to OS working set from the cache. Thus, user cache hit-rates are maximized at $N = 0$.

The OS and application do not have completely independent working sets – much of their memory footprint is shared. This sharing occurs because the OS often performs operations such as I/O on behalf of the application and places the

resulting data into the application address space. As a result, shared data that is written, can generate substantial coherence traffic between OS and User cores. As we move to low values for N, coherence traffic is also maximized because OS/User shared data is no longer in just a single cache.

For example, Figure 4 shows that even with a zero overhead off-loading latency, moving from $N = 100$ to $N = 0$ substantially reduces performance. This is because the cost of additional coherence invalidates and transfers overshadows any improvements in cache hit-rate. Because the relationship between coherence and cache interference can vary from application to application, it is not possible to globally determine the point N at which off-loading performance is maximized. A dynamically adjusting system utilizing performance feedback is required.

Off-loading short OS sequences is required. We are somewhat surprised to see that maximum performance occurs when off-loading OS invocations as short as 100 instructions long. This indicates that though long duration invocations dominate the total OS instruction count, short frequent invocations also have a large impact on cache interference for some benchmarks. This implies that any software-based decision policy that relies on OS code instrumentation must not only consider long-running system calls, but also short-running ones. This supports our belief that all OS entry points must be instrumented for off-loading, and in turn, that low-cost instrumentation, such as the hardware predictor presented here, is necessary for OS off-loading to be successful.

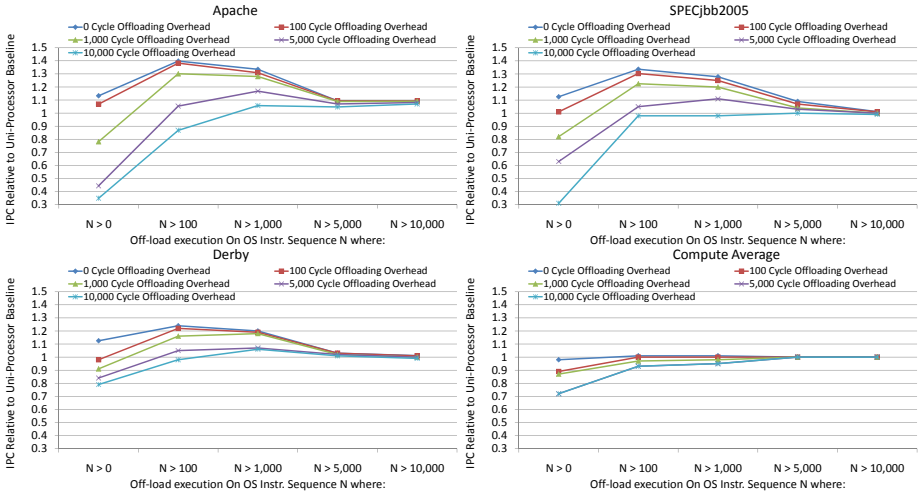

Fig. 4. Normalized IPC relative to uni-processor baseline when varying the off-loading overhead and the switch trigger threshold.

5.2 Comparing Instrumentation and Hardware Prediction

Having studied the behavior of the hardware predictor, we now compare our eventual hardware-based scheme against existing techniques that have relied on manual software instrumentation. Figure 5 provides a concise view of what we consider to be state of the art in OS off-loading. We present normalized throughput results for an off-loading latency of 5,000 cycles (Conservative), that is currently available, as well as 100 cycles (Aggressive) that has been proposed in research. Static instrumentation (SI) models a low overhead software instrumentation that uses off-line profiling to identify and statically instrument only those OS routines that are determined to have a run-length that is twice the off-loading (migration) latency. SI is very similar to the technique used by Chakraborty et al. [10]. Dynamic instrumentation (DI) models a more complex software instrumentation of all possible OS entry points and allows run-time decisions about the value of off-loading. DI is very similar to the technique used by Mogul et al. [17], but differs in that all OS entry points are instrumented, as opposed to just those which are expected to have a long average run length. DI is the functional equivalent of the hardware prediction engine proposed in this paper, but implemented entirely in software. Finally, hardware instrumentation (HI) is the new hardware predictor based approach described in this work.

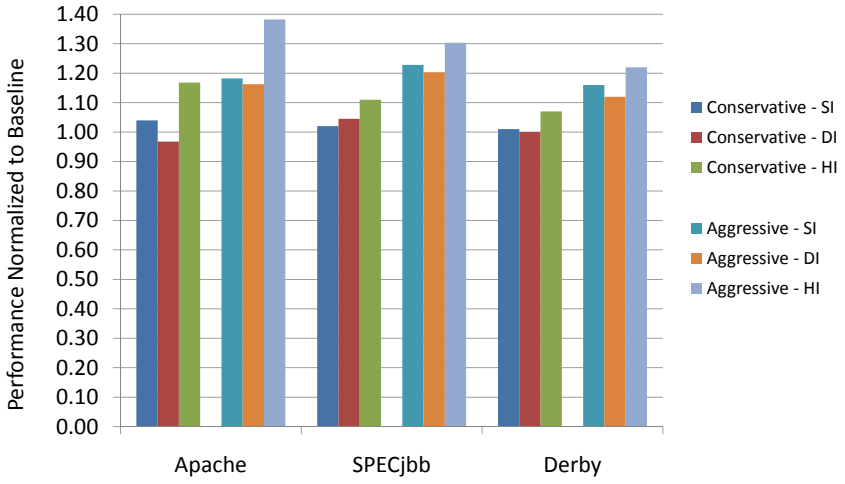

Fig. 5. Normalized throughput (relative to a single-core baseline) for off-loading with static manual instrumentation and off-loading with the hardware predictor.

Figure 5 indicates that previous OS off-loading proposals were leaving substantial performance on the table by (i) not considering short OS sequences as candidates for off-loading, and (ii) utilizing high overhead software instrumentation to make off-loading decisions. At currently achievable off-loading latencies, our hardware based off-loading decision policy can yield as much as 18%

throughput improvement over a baseline without off-loading, and 13% improvement over previous software-based implementations. In the future, with faster migration implementations, the importance of a low overhead decision making policy will increase and our proposal can outperform software instrumentation by as much as 20%.

In our experiments, the single-core baseline has a single 1 MB L2 cache, while the off-loading models have two 1 MB L2 caches. This is a faithful model of how off-loading is expected to work in future multi-cores, and the additional cache space is a strong contributor to the high benefit from off-loading. It is worth noting that even an off-loading model with two 512 KB L2 caches can outperform the single-core baseline with a 1 MB L2 cache if the off-loading latency is under 1,000 cycles. Such a comparison only has academic value however, as hardware designers are unlikely to cripple an existing processor design to enable off-loading.

5.3 Scalability of Off-loading

Table 3 shows the percentage of benchmark execution time that the OS core was active while running our server intensive benchmarks with a 5,000 cycle off-loading overhead. At an off-loading threshold of $N = 10,000$ the OS core shows low utilization, but at $N = 1000$ and $N = 100$, the threshold where optimal performance is found for most benchmarks, the utilization quickly increases. This high rate of utilization indicates that, unfortunately, it is unlikely that multiple user-cores will be able to share a single OS core successfully.

Table 3. Percentage of total execution time spent on OS-Core using selective migration based on threshold N

	Core Migration Threshold N			
Benchmark	**100**	**1,000**	**5,000**	**10,000+**
Apache	45.75%	37.96%	17.83%	17.68%
SPECjbb2005	34.48%	33.15%	21.28%	14.79%
Derby	8.2%	5.4%	1.2%	0.2%

To test this hypothesis we evaluated the scaling of a single OS core to two and four user cores using SPECjbb2005, under an off-loading threshold of $N = 100$ and an aggressive off-loading overhead of 1,000 cycles. As a non-SMT core, if the OS core is handling an off-loading request when an additional request comes in, the new request must be stalled until the OS core becomes free. With just two user cores, there was an average queuing delay of 1,348 cycles in addition to the 1,000 cycle off-loading overhead. With four user cores, the average queuing delay for the OS core exploded to over 25,000 cycles as the OS core was inundated with off-loading requests. While L2 cache hit rates remained high in both scenarios, our user cores were often stalled waiting for the OS off-loading to occur. As a result, the 2:1 ratio of User to OS cores saw only a 4.5% improvement in aggregate throughput, and performance was decreased substantially at the 4:1

ratio. We conclude that 1:1, or possibly 1:N, may be the appropriate ratio of provisioning OS cores in a many-core system.

5.4 TLB Impact

Off-loading OS execution to a secondary core impacts the utilization of the TLB. Compared to the single-core baseline, OS off-loading causes a subset of the application's memory footprint to be accessed by either the user core, OS core, or both. Figure 6 shows the baseline TLB hit-rate, as well as the TLB hit-rate in the maximal off-loading (N=0) situation. We find that by segregating OS and User references, the TLB hit-rate for both OS and User cores improves when each core maintains a full size 128 entry TLB. Thus off-loading effectively extends the reach of the TLB compared to a single-core baseline. The hit-rate for reduced size TLBs are also shown and indicate that the OS core is particularly sensitive to any reduction in TLB size. User core TLB hit-rates out perform the baseline TLB even when reduced to half the baseline size. TLB behavior is often critical to performance, and OS-off-loading provides a mechanism to improve TLB hit-rates without increasing the size (and thus latency) of the TLB, a structure that is often on a critical timing path.

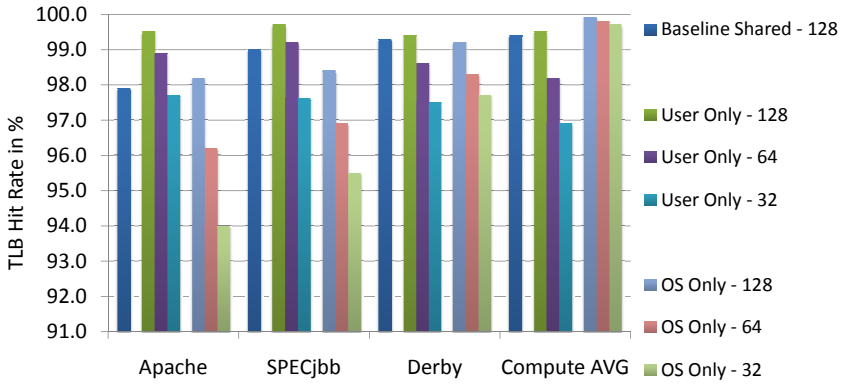

Fig. 6. TLB Hitrates Of User/OS Core When Off-loading All Available OS Sequences

6 Related Work

6.1 Impact of OS on System Throughput

There have been many studies on how operating system overhead affects the throughput of user applications (the eventual metric of interest). Gloy et al. [11], Anderson et al. [4], and Agarwal et al. [2] 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. [19]

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. [19] and Li et al. [15] show OS execution under-performs user applications by 3-5x on modern out-of-order processors and suggest that OS code can be run on less aggressively designed processors to improve energy efficiency.

6.2 Hardware Support for Efficient OS Execution

Several groups have proposed that a class of OS intensive workloads combined with the proliferation of chip multiprocessors has led us to an architectural inflection point, where off-loading operating system execution may be beneficial for both performance and power efficiency. As already described, Chakraborty et al. [10] have proposed that some system calls should be selectively migrated to and executed on an alternate, homogeneous core within a CMP. This results in better cache locality in all cores, yielding higher server throughput without requiring additional transistors.

Mogul et al. [17] recently proposed that some OS system calls should selectively be migrated to and executed on a microprocessor with a less aggressive microarchitecture. OS code does not leverage aggressive speculation and deep pipelines, so the power required to implement these features results in little performance advantage. While system calls are executing on the low-power OS core, the aggressively designed user core can enter a low-power state. When the OS routine has completed execution, the user core is powered back up and execution returns to the high performance user core while the OS core enters low-power state. This migration of OS execution to an energy-efficient core results in overall lower energy for the system.

Li et al. [15] take a slightly different approach to OS execution than other proposals. They propose that rather than implementing a secondary processor, existing uni-processors should be augmented so that aggressive out-of-order features can be throttled in the microarchitecture. By limiting the instruction window and issue width, they are able to save power during operating system execution. Similar to previous proposals, they still must identify the appropriate opportunities for microarchitectural reconfiguration to occur. Identification of reconfiguration opportunities has many of the same steps as off-loading identification. We believe our hardware-based decision engine could be utilized effectively for the type of reconfiguration proposed by Li et al.

7 Conclusions

Off-load of OS functionality has promise in future multi-cores because it can afford better cache utilization and energy efficiency. While it has traditionally been assumed that off-loading only makes sense for long OS executions, we show that the off-load of short sequences also contributes greatly to reduced cache interference. Prior work has implemented off-loading with profile-guided software instrumentation. We show that such an approach is burdensome, incurs high

overheads (especially when off-loading short sequences), and is often inaccurate. All of these problems are addressed by instead implementing a hardware tunable predictor that estimates the length of OS sequences and off-loads upon expected benefit. The predictor has a storage overhead of only 2 KB and out-performs the best instrumentation policy by 13%. For future work, we plan to study the applicability of the predictor for OS energy optimizations.

References

1. The SPARC Architecture Manual Version 9, http://www.sparc.org/standards/SPARCV9.pdf
2. Agarwal, A., Hennessy, J., Horowitz, M.: Cache Performance of Operating System and Multiprogramming Workloads. ACM Trans. Comput. Syst. 6(4), 393–431 (1988)
3. Albayraktaroglu, K., Jaleel, A., Wu, X., Franklin, M., Jacob, B., Tseng, C.W., Yeung, D.: BioBench: A Benchmark Suite of Bioinformatics Applications. In: Proceedings of ISPASS (2005)
4. Anderson, T.E., Levy, H.M., Bershad, B.N., Lazowska, E.D.: The Interaction of Architecture and Operating System Design. In: Proceedings of ASPLOS (1991)
5. Balasubramonian, R., Dwarkadas, S., Albonesi, D.: Dynamically Managing the Communication-Parallelism Trade-Off in Future Clustered Processors. In: Proceedings of ISCA-30. pp. 275–286 (June 2003)
6. Barroso, L., Holzle, U.: The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan & Claypool (2009)
7. Baumann, A., Barham, P., Dagand, P., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schupbach, A., Singhania, A.: The Multikernel: A new OS architecture for scalable multicore systems. In: Proceedings of SOSP (October 2009)
8. Benia, C., et al.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. Tech. rep., Department of Computer Science, Princeton University (2008)
9. Brown, J.A., Tullsen, D.M.: The Shared-Thread Multiprocessor. In: Proceedings of ICS (2008)
10. Chakraborty, K., Wells, P.M., Sohi, G.S.: Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-Fly. In: Proceedings of ASPLOS (2006)
11. Gloy, N., Young, C., Chen, J.B., Smith, M.D.: An Analysis of Dynamic Branch Prediction Schemes on System Workloads. In: Proceedings of ISCA (1996)
12. Henning, J.L.: SPEC CPU2006 Benchmark Descriptions. In: Proceedings of ACM SIGARCH Computer Architecture News (2005)
13. Hunt, G., Larus, J.: Singularity: rethinking the software stack. In: Operating Systems Review (April 2007)
14. Li, T., John, L., Sivasubramaniam, A., Vijaykrishnan, N., Rubio, J.: Understanding and Improving Operating System Effects in Control Flow Prediction. Operating Systems Review (December 2002)
15. Li, T., John, L.K.: Operating System Power Minimization through Run-time Processor Resource Adaptation. IEEE Microprocessors and Microsystems 30, 189–198 (June 2006)
16. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. IEEE Computer 35(2), 50–58 (February 2002)

17. Mogul, J., Mudigonda, J., Binkert, N., Ranganathan, P., Talwar, V.: Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. IEEE Micro (May-June 2008)
18. Muralimanohar, N., Balasubramonian, R., Jouppi, N.: Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In: Proceedings of MICRO (2007)
19. Nellans, D., Balasubramonian, R., Brunvand, E.: A Case for Increased Operating System Support in Chip Multi-Processors. In: Proceedings of the 2nd IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (September 2005)
20. Nellans, D., Balasubramonian, R., Brunvand, E.: OS Execution on Multi-Cores: Is Out-Sourcing Worthwhile? ACM Operating System Review (April 2009)
21. Redstone, J., Eggers, S.J., Levy, H.M.: An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In: Proceedings of ASPLOS (2000)
22. Strong, R., Mudigonda, J., Mogul, J., Binkert, N., Tullsen, D.: Fast Switching of Threads Between Cores. Operating Systems Review (April 2009)
23. U.S. Environmental Protection Agency - Energy Star Program: Report To Congress on Server and Data Center Energy Efficiency - Public Law 109-431 (2007)