IMPROVING OPERATING SYSTEM AND HARDWARE INTERACTIONS THROUGH CO-DESIGN

by

David Nellans

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2014

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **David Nellans**

has been approved by the following supervisory committee members:

Erik Brunvand	, Chair	**2-May-2014** Date Approved
Rajeev Balasubramonian	, Member	**2-May-2014** Date Approved
Alan Davis	, Member	**2-May-2014** Date Approved
John Carter	, Member	**1-Oct-2014** Date Approved
Jeffrey Mogul	, Member	**16-Sep-2014** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

With the explosion of chip transistor counts, the semiconductor industry has struggled with ways to continue scaling computing performance in line with historical trends. In recent years, the de facto solution to utilize excess transistors has been to increase the size of the on-chip data cache, allowing fast access to an increased portion of main memory. These large caches allowed the continued scaling of single thread performance, which had not yet reached the limit of instruction level parallelism (ILP). As we approach the potential limits of parallelism within a single threaded application, new approaches such as chip multiprocessors (CMP) have become popular for scaling performance utilizing thread level parallelism (TLP).

This dissertation identifies the operating system as a ubiquitous area where single threaded performance and multithreaded performance have often been ignored by computer architects. We propose that novel hardware and OS co-design has the potential to significantly improve current chip multiprocessor designs, enabling increased performance and improved power efficiency. We show that the operating system contributes a nontrivial overhead to even the most computationally intense workloads and that this OS contribution grows to a significant fraction of total instructions when executing several common applications found in the datacenter. We demonstrate that architectural improvements have had little to no effect on the performance of the OS over the last 15 years, leaving ample room for improvements.

We specifically consider three potential solutions to improve OS execution on modern processors. First, we consider the potential of a separate operating system processor (OSP) operating concurrently with general purpose processors (GPP) in a chip multiprocessor organization, with several specialized structures acting as efficient conduits between these processors. Second, we consider the potential of segregating existing caching structures to decrease cache interference between the OS and appli-

cation. Third, we propose that there are components within the OS itself that should be refactored to be both multithreaded and cache topology aware, which in turn, improves the performance and scalability of many-threaded applications.

Thanks Theo.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thanks to everyone at Utah, Sun, Utah, Intel, Utah, Prediction, more Utah, Fusion-io, Nvidia, and Utah yet again, for getting me to finish despite my numerous distractions.

CHAPTER 1

INTRODUCTION

The performance of computer systems has scaled well due to a synergistic combination of technological advancement and architectural improvement. In the last 15 years, high performance computing has progressed from the venerable single core 486/33, released in 1989, to the current IBM Power 8 and many-core Intel Xeon series. Process fabrication size has shrunk from 1 micron (486/33) down to 32 nanometers (Xeon Sandybridge) and is expected to continue down to 22 nanometer and below. Dramatically decreased transistor sizes have helped enable a 100-fold increase in clock frequency from 33MHz to 3.8GHz during this same period. Simultaneously, the number of pipeline stages has increased from the classic 5 stage pipeline all the way up to 31 stages [1] in search of ILP but has recently relaxed to a more moderate 12 stages in an effort to find a balance between absolute performance and power efficiency. Transistor count and resultant die size has exploded from 1.2 million transistors on the 486/33 to several billion in complex chip multiprocessors.

At the same time as technology has improved, architectural improvements such as deep pipelines, caching, and out of order execution have allowed us to take advantage of increased transistor counts to improve system performance at a breakneck pace. If we decompose the performance improvements from the 486 to the P4 (often regarded as one of the most aggressively designed cores for extracting ILP), general application performance has improved approximately 200x in this time period, yet we find that the performance of the operating system in the same period has seen a significantly smaller improvement on the order of 50x. For domains where the operating system contributes a significant percentage of cycles, this can have a serious impact on overall system performance.

Many believe we are approaching a performance limit in single threads due to available instruction level parallelism (ILP) [2, 3]. Methods to extract ILP are well known but the power that must be expended to mine this ILP within single-threaded programs is becoming overwhelming. Power dissipation has traditionally been dominated by dynamic switching but as process sizes decrease leakage power also becomes a significant issue. Techniques exist to decrease both static and dynamic power dissipation such as frequency scaling, clock gating, voltage scaling, and power gating [4–8]. None of these techniques can be applied without disrupting the normal pipeline flow of the microprocessor and require stalling the pipeline anywhere from a handful of cycles in frequency scaling to a several of microseconds in power gating for circuits to stabilize. The high overhead for aggressive techniques such as power gating can often limit the usefulness of these methods, though several research and product groups are actively working to reduce these costs [9–12].

To achieve increased performance within limited power budgets, designers are turning toward thread level parallelism (TLP) to increase system throughput. Simultaneous multithreading (SMT) [13] increases the utilization of microarchitectural structures by dynamically sharing resources between two or more active threads. Chip multiprocessing (CMP) [14, 15] statically partitions these resources providing multiple, possibly simpler, copies of a processing core on a single die. Both of these methods have little benefit for single thread performance but can significantly increase total system throughput for highly threaded workloads such as database and web servers [16]. SMT and CMP technologies complement each other well; IBM and Intel both have released processors which concurrently utilize both technologies.

Modern HPC environments are moving towards designs that are symmetric multiprocessors designs of chip multiprocessors, which are themselves using simultaneous multithreading. The end result is a cache coherent, but extremely complex, system with very complex cache coherence mechanisms and private and shared cache designs.

One contributor to the lack of performance scaling for OS intensive workloads is that the operating system is simply not typically considered when architectural enhancements are proposed for general purpose processors. Modern processors, especially in an academic setting, are typically simulated using a cycle accurate ar-

chitectural model, such as Simplescalar [17], to evaluate performance. Typically these cycle accurate simulators do not execute the operating system because doing so requires accurate I/O models and slows down simulation speed substantially. Full system simulators are becoming more prevalent [18–21] but often require modification to the operating system or do not provide cycle accurate numbers. While several studies suggest that the operating system has significant role in commercial workload performance [16, 22–24], the effect of the operating system on varying workloads is still largely unexplored.

1.1 Problem Approach

We approach the problem from two distinct angles. First we intend to use current *real* hardware to measure the actual effects of the operating system on overall application performance. This approach to categorizing operating system effects has not been explored extensively to our knowledge. This area is largely unexplored because there is a lack of tools and operating system integration to be able to isolate operating system functionality at a resolution that is meaningful from a performance analysis standpoint. Until recently most commercial microprocessors also were not, or were not exposing, performance analysis circuitry to the system programmer. This meant that even if you could profile your application and operating system code at a fine grained resolution, the resolution of timers and performance counting events were not sufficient to gain meaningful results. Often timing resolution was limited to milliseconds, which is far too large a time frame for computer architects to profile what is happening within a microprocessor which is performing hundreds of thousands of events.

In late 2005, both operating systems and microprocessors have been able to expose a rich set of events and performance counters, which allow accurate accounting of event costs. This allows us to profile operating system events and their effect on the user executed code as well. This approach of determining performance costs and frequencies is necessary for this work because of the lack of operating system support in most architectural simulators. While operating system support is gaining momentum, the complexity required to fully support, and be able to track, operating

system events is tremendous. Currently there are two simulators available that allow operating system execution as part of microprocessor and system characterization and design. Simics [25] is a commercial simulator produced by Wind River, and Gem5 [26] is available within the academic community but is starting to have use in industrial research laboratories as well. While both simulators allow operating system execution, the fidelity of the microarchitectural models they chose to implement can vary dramatically. For example, neither simulator supports an accurate multisocket topology using an packet based interconnect such as Intel's QPI or AMD's Hypertransport.

While we utilize real hardware to determine the actual cost of operating system events, you can not modify existing hardware to explore architectural advancements. Thus, based on our real world numbers obtained in the first step, we implement a chip multiprocessor design using architectural simulation that is modified for efficient operating system execution. Within this simulation we can test architectural features of the operating system processor such as modifying or specializing functional units, reducing pipeline length, and how interrupt handling occurs. These features can only be implemented at the level of detail supported by the simulator, so while some features may be implemented at a cycle accurate level, others may have to be implemented using a black box timing model. We implemented as much as possible at the finest level of detail and only utilized the black box timing model when necessary basing our timing assumptions on the real world timings obtained via our first approach.

The simulation based approach encourages exploration of hardware optimizations to support operating system execution at the microscopic level. An alternative approach we also explore is to modify the operating system to more intelligently make use of the existing hardware available today. When this work was started, chip multiprocessors were just coming on the scene and two-core boxes were prevalent. There are now 8-socket, 80-core, and 160 logical processor machines available from most large OEM vendors, making OS changes that run on real hardware a viable problem approach necessary to explore.

1.2 Thesis Statement

This dissertation tests the hypothesis that intelligent use and co-design of operating system and hardware systems can improve overall system performance and power efficiency for full system applications; it demonstrates three different possibilities for improvement. i) where an operating system processor is used to offload execution under hardware control, improving the on-chip cache efficiency of the system. ii) where a single traditional CPU cache is made OS-aware and segregates OS and application references so they do not displace each other greedily. iii) where OS execution is redesigned to be CPU and cache organization aware and can dynamically adapt to different system configurations for optimal performance requiring no hardware modifications.

1.3 Organization

This dissertation is organized into seven chapters. Chapter 2 provides background on why OS intensive applications have not scaled in performance over the last 20 years, and Chapter 3 provides an overview of work that motivated and relates to this dissertation. Each subsequent chapter describes mechanisms to improve OS and overall system, performance, and power consumption. Chapter 4 describes techniques for offloading OS execution to a OS coprocessor without operating system intervention. Chapter 5 provides an alternative to OS offloading which is to segregate multiway caches between the OS and application at runtime. Chapter 6 provides a mechanism to isolate the OS on a modern SMP of CMP class machine and provides data on what is possible with a modern machine without requiring any hardware modifications. Chapter 7 finally summarizes the conclusions and describes future directions that may result from this work.

CHAPTER 2

BACKGROUND AND MOTIVATION

Computationally intense workloads are used to evaluate microarchitectural improvements because they maximize computation and minimize the effects of memory and I/O performance as much possible. These benchmarks often represent scientific workloads that require minimal user intervention. These benchmarks are ideal for determining the minimal amount of operating system overhead that will be present on a machine because they make very small use of operating system functionality. Thus overhead is likely to be contributed from timer interrupts resulting in context switching, interrupt handling, and system daemon overhead.

We chose to use SPECint, SPECfp, and ByteMark [27] as our computationally intense benchmarks. We also chose to time a Linux kernel compile to provide a computationally intense workload that also provides some file system I/O. Table 2.1 shows the percentage of instructions executed in supervisor mode by these benchmarks. Table 2.2 shows the individual breakdown and variation between the SPEC benchmarks. SPEC benchmarks, particularly those requiring Fortran, for which simulation was unable to complete due to unidentifiable internal Simics errors, are not shown. The average operating system overhead when executing these four benchmarks is 9.43% of total instructions executed. The ByteMark benchmark skews these results strongly, however, and we believe the average of 5.19% of instructions for the SPEC benchmarks is a more realistic minimal overhead. For the purpose of this work, computationally intense workloads are not the target applications, but we include them for a reference to know if our OS specific optimization may end up hurting this broad class of applications that are important nevertheless.

Computationally intense benchmarks are a good way to test architectural im-

Table 2.1. Operating system contribution in various workloads on Pentium 4 Prescott 90nm.

Benchmark	% Instructions contributed by OS
Linux Kernel Compile	3.63%
SPECint Average	5.15%
SPECfp Average	5.24%
ByteMark	23.73%
Bonnie++	86.41%
UnixBench	97.16%
NetPerf	99.48%
X Usage	38.04%
Average	44.85%

provements in the architecture but rarely do they capture total system performance because many applications involve a significant amount of file or network traffic. We chose to use Bonnie++, an open source file-system performance benchmark [28] and netperf, a TCP performance benchmark, to measure OS contribution in what we expected to be OS dominated workloads. Table 2.1 shows that our expectations were indeed correct and that the OS contribution far outweighs user code contribution for file-system and network operations. We also used UnixBench as an I/O Intensive workload. UnixBench consists of several benchmarks including Whetstone, Drystone, system call performance monitoring, and typical UNIX command line usage.

UnixBench has a very high operating system contribution of 97.16% as it is meant to measure operating system performance through a variety of microbenchmarks measuring various OS subsystem performance. These benchmarks confirm that I/O processing is almost entirely OS dominated and support the work of Redstone et al. [16] who have shown that for workloads such as Apache the operating system can contribute more than 70% of the total cycles. Userland applications that contain a mix of floating point and integer operations tend to approach a maximum CPI of 2.0 with perfect instructions mix, while integer-only benchmarks such as specCPU2006 have an IPC of just 1.0. Because operating system code is typically entirely integer based instructions, even going so far as to emulate floating point with integer, in some cases for compatibility, the maximum IPC that is achievable is typically 1.0 in

Table 2.2. SPECcpu2000 instructions executed in supervisor mode on Pentium 4 Prescott 90nm.

SPECfp	**Instructions**	**SPECint**	**Instructions**
Wupwise	5.96%	Gzip	4.65%
Swim	19.42%	Vpr	4.43%
Mgrid	0.59%	Gcc	5.04%
Applu	4.29%	Mcf	4.86%
Mesa	0.71%	Crafty	4.54%
Galgel	0.51%	Parser	4.82%
		Eon	4.58%
		Gap	5.91%
		Vortex	6.62%
		Bzip2	6.32%
		Twolf	4.97%
Average	5.24%	Average	5.15%

practice for tight cache resident code. The applications we look at in this dissertation, interactive aside, are dominated by integer operations almost exclusively.

While CPU-intensive benchmarks provide useful data when measuring processor innovations, these workloads rarely represent the day to day usage of many computers throughout the world. One key difference between these workloads and typical workstation use is the lack of interaction between the application and the user. X Windows, keyboard, and mouse use generates a workload that is very interrupt driven even when the user applications require very little computation, a common case within the consumer desktop domain. When a high performance workstation is being utilized fully, either locally or as part of a distributed cluster, the processor must still handle these frequent interrupts from user interaction, thus slowing down the application computation.

We modeled an interactive workload by simultaneously decoding an MP3, browsing the web and composing email in a text editor. While this type of benchmark is not deterministically reproducible across multiple runs due to spurious interrupt timing and differences in workload actions being performed by a user, we believe smoothing the operating system effects across many billions of instructions and many minutes of wall clock time accurately portrays the operating system contribution.

This interactive workload typically only utilized 33% of the processor's cycles and spent 38% of these instructions within the OS.

2.1 Decomposing Performance Improvements

We have looked at several areas of computer performance to determine the amount of improvement operating systems have achieved in the last 15 years when compared to user-land applications. We look at time spent in OS code on behalf of the user application, context switch time, and system call performance.

2.1.1 Operating System Performance

The performance increase in microprocessors over the past 15 years has come from both architectural improvements as well as technological improvements. Faster transistors have helped drive architectural improvements such as deep pipelining, which in turn caused an enormous increase in clock frequencies. Significant changes have occurred in architecture as well, moving from the classic five stage pipeline to a multiple issue, deeply pipelined architecture, where significant resources are utilized in branch prediction and caching. Because we wish to distill the architectural improvement in the last 15 years and disregard technological improvement as much as possible, we chose to take measurements from real machines, a 486 @ 33MHz and a Pentium 4 @ 3.0GHz, instead of relying on architectural simulations which can introduce error due to inaccurate modeling of the operating system. For this purpose we define total performance improvement (P) as technology improvement (T) times architectural improvement (A), or $P = T \times A$.

A metric commonly used to compare raw circuit performance in different technologies is the fan-out of four (FO4) [29, 30]. This is defined to be the speed at which a single inverter can drive four copies of itself. This metric scales roughly linearly with process feature size. Thus a processor scaled from a 1 micron process, our 486, to 90nm, our Pentium 4, would have approximately an 11-fold decrease in FO4 delay. We set T, our technological improvement, to 11 for the remainder of our calculations.

Table 2.3 shows the performance difference when executing the same workload on both machines using the UNIX time command. The total performance improvement P is taken by examining the *Real* time. We found the average speedup, including both

Table 2.3. Speedup from 486 33MHz to Pentium 4 3.0GHz Prescott 90nm.

Benchmark	486	Pentium 4	Speedup
crafty	Real:27,705s	Real: 115s	240.91
(SPECint)	User:27,612s	User: 110s	251.01
	Sys: 14s	Sys: 1s	14.00
twolf	Real:35,792s	Real: 249s	143.74
(SPECint)	User:35,191s	User: 234s	150.38
	Sys: 49s	Sys: 1s	49.00
mesa	Real:51,447s	Real: 272s	189.14
(SPECfp)	User:50,801s	User: 249s	204.02
	Sys: 112s	Sys: 2s	56
art	Real:64,401s	Real: 332s	193.97
(SPECfp)	User:64,160s	User: 306s	209.67
	Sys: 34s	Sys: 1s	34
Linux Kernel	Real:57,427s	Real: 292s	196.66
Compile	User:54,545s	User: 250s	218.18
	Sys: 1,930s	Sys: 25s	77.2

application and operating system effects, when moving from the 486 to the Pentium 4 was 192.2. Using 192.2 and 11, for P and T, respectively, we can then calculate that our architectural improvement, A, is 17.47 or roughly 60% more improvement than we have obtained from technological improvements. This architectural improvement comes from increased ILP utilization due to branch prediction, deeper pipelines, out of order issue, and other features not present in a classic five stage pipeline.

2.1.2 System Call Performance

To further validate our results that the OS performs anywhere from 4–10 times worse than user codes on modern architectures we used the Linux Trace Toolkit (LTT) [31] to log system call timings over a duration of 10 seconds on both the 486 and the Pentium 4. LTT is commonly used to help profile both applications and the operating system to determine critical sections that need optimization. LTT can provide the cumulative time spent in each system call as well as the number of invocations during a tracked period of time. This allows us to average the execution of system calls over thousands of calls to minimize variation due to caching, memory, and disk performance. By averaging these 10 second runs across the multiple workloads

found in Table 2.4 we also eliminate variation due to particular workload patterns.

The Linux Trace Toolkit provides microsecond timing fidelity; thus system calls that take less than 1 microsecond can be reported as taking either 0 microseconds or 1 microsecond depending on where the call falls in a clock period. Averaging thousands of calls to such routines should result in a random distribution across a clock period, but we currently have no way to measure if this is true and thus can not guarantee if this is, or is not, occurring. All system call timings have been rounded to the nearest microsecond.

When examining these figures we must be careful to examine the function of the system call before interpreting the results. System calls that are waiting on I/O such as poll or are waiting indefinitely for a signal such as waitpid should be disregarded because they depend on factors outside of OS performance. System calls such as execve, munmap, and newuname provide more accurate reflections of operating system performance independent of device speed. Because system calls can vary in execution so greatly, we do not attempt to discern an average number for system call speedup at this time, instead providing a large number of calls to examine. It is clear, however, that most system calls in the operating system are not gaining the same benefit from architectural improvement, 17.47, that user codes have received in the past 15 years.

2.1.3 Context Switch Performance

Figure 2.1 provides the average context switch time, normalized to 3.0 GHz, over five runs of our benchmark on various machine configurations. All timings showed a standard deviation of less than 3% within the five runs. The absolute number for context switch time is much less important than the relative time between cases. Disregarding the overhead of the token passing method lets us focus on the relative change in context switch cost between differing machine architectures. Context switching routines in the Linux kernel are hand-tuned sections of assembly code that do not make use of the hardware context switch provided by the x86 instruction set. It has been shown that this software context switch has performance comparable to the hardware switch but provides more flexibility.

Our first experiment sought to determine how context switch time scaled with

Table 2.4. System call speedup from 486 33MHz to Pentium 4 3.0GHz Pentium 4 Prescott 90nm using identical Linux kernel version 2.6.28. (System call times in microseconds. Speedup - greater than zero denotes improvement, less that zero denotes wall clock reduction in performance.)

System Call	486	P4	Speedup	Arch Speedup
brk	87	2	43	3.90
close	439	29	15	1.36
execve	14,406	1,954	7	0.63
fcntl64	62	1	62	5.63
fork	15,985	8,187	2	0.18
fstat64	183	1	183	16.63
getdents64	501	10	50	4.54
getpid	49	1	49	4.45
getrlimit	59	1	59	5.36
ioctl	728	25	29	2.63
mprotect	324	4	81	7.36
munmap	365	11	33	2.99
newuname	88	2	44	43.99
open	860	899	< 0	< 0
pipe	559	7	79	7.18
poll	9,727,367	9,280	1,048	95.27
read	44,304	185	239	21.72
rt_sigaction	206	1	206	18.72
rt_sigprocmask	178	1	178	16.18
select	403,042	11,849	34	3.09
sigreturn	75	1	76	6.90
stat64	264	53	5	0.45
time	72	5,585	<0	<0
waitpid	99,917	3	33,305	3027.72
write	2,164	3,418	<0	<0

clock frequency for a given architecture. Our Pentium 4 running at 3.0GHz supports frequency scaling allowing us to scale down the frequency of the processor in hardware and run our context switching benchmark at multiple frequencies on the same processor. The absolute context switch time for these frequency scaled runs was then *normalized back to 3.0Ghz*. This normalization allows us to clearly see that context switch time scales proportionally with clock frequency. Scaling proportionally with clock frequency indicates that context switching is a fairly memory independent

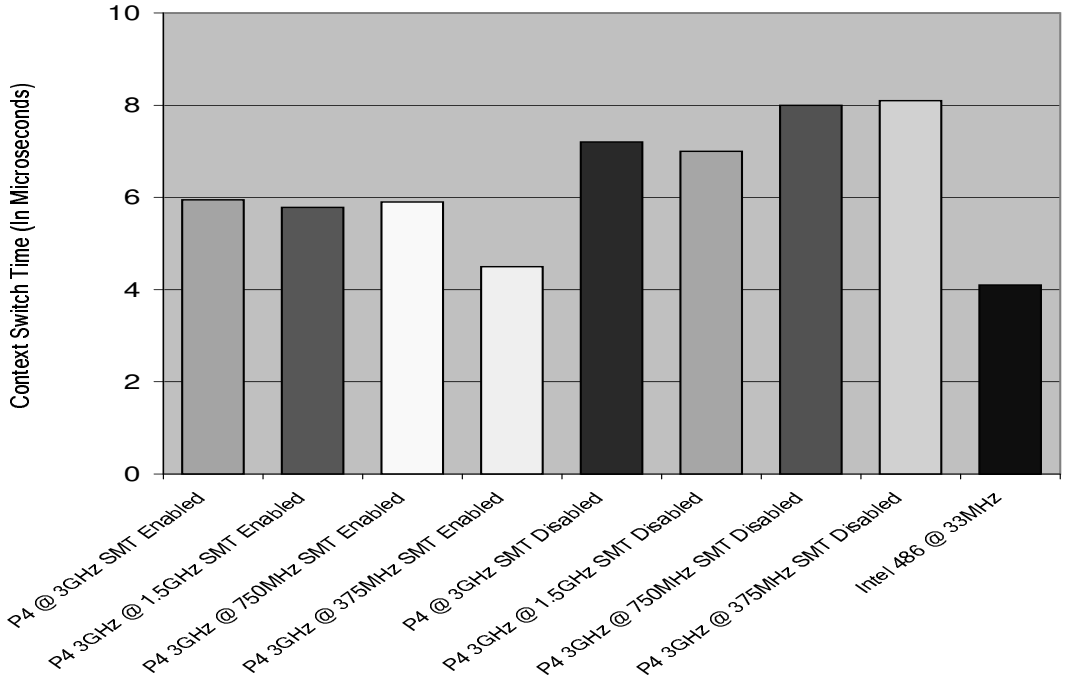

Figure 2.1. Context switching speed normalized for clock frequency.

operation. Thus we draw the conclusion that the number of cycles required to context switch in a given microarchitecture is independent of clock frequency as it requires a pipeline flush which dominates the execution time.

To determine if architectural improvements have reduced context switch time we run this benchmark on a 486 @ 33MHz with the identical kernel and tool-chain versions. Scaled for frequency, Figure 2.1 shows that context switch performance has actually *decreased* in the last 15 years by requiring an *increased* percentage of cycles. This is likely due to the increased cost of flushing a modern Pentium's 31 pipeline stages versus the classic five stages in a 486. This decrease in context switch performance undoubtedly contributes to the lackluster performance of the operating system that we have observed.

2.1.4 Interrupt Handling

Under the Linux OS the external timer interval is by default 10ms. This provides the maximum possible idle time for a processor running the operating system. Including external device interrupts, the interval between interrupts dropped to 6.2ms on average for both the 486 and the Pentium 4. Each interrupt causes on average 1.2

context switches and requires 3 microseconds to be handled. The irq handling cost is negligible compared to the context switching cost and can be disregarded. Thus the average cost of handling 193 context switches at approximately 18,000 cycles per context switch, 6 millisecond context switch time measured on a 3GHz P4, requires 3.5 million cycles per second, or only about 0.1% of the machines cycles.

While the total number of cycles spent handling interrupts is very low, the performance implications are actually quite high. The regular nature of interrupt handling shown in Figure 2.2 generates regular context switching, which in turn causes destructive interference in microarchitectural structures such as caches and branch prediction history tables. The required warm up of such structures on every context switch has been shown to have significant impact on both operating system and application performance [32].

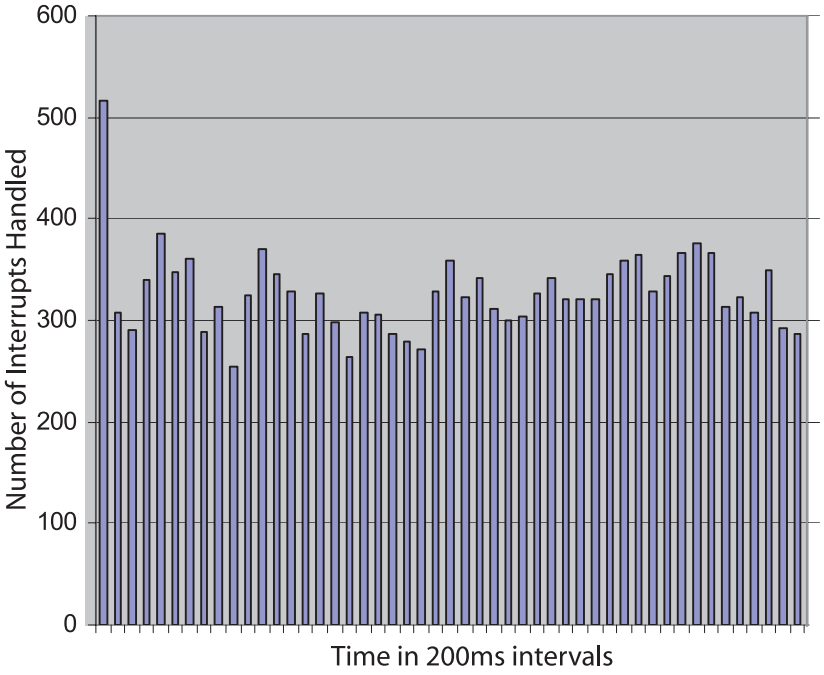

Figure 2.2. Distribution of interrupt handling over 10 seconds.

CHAPTER 3

RELATED WORK

There have been many pieces of work, both historical and recently, which contribute to the general body of work relating to measuring and improving the effects of operating system execution. In addition to the related work cited in-line as part of this dissertation, we present a short summary of selected works that are closely related to this dissertation because they have provided much insight into the potential benefits of separating the operating system processor both logically and physically in a chip multiprocessor.

3.1 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. [33], Anderson et al. [34], and Agarwal et al. [35] have shown that operating system execution generates memory references that negatively impact the performance of traditional memory hierarchies. Redstone et al. [16] and Nellans et al. [36] 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. [36] and Li et al. [37] show OS execution underperforms 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.

3.2 Hardware Support for 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 offloading operating system execution may be beneficial for both performance and power efficiency.

3.2.1 Offloading for Performance

Chakraborty et al. [38] have proposed that some system calls should be selectively migrated to and executed on an alternate, homogeneous, microprocessor within a chip multiprocessor design. By migrating OS execution away from the user processor, there is less contention for private cache resources between the OS and user code, resulting in better cache locality for both. In turn, this speeds up execution and can overcome the latency required to migrate execution to an alternate processor. They show that for server-oriented workloads it is possible to improve system throughput without requiring additional coprocessors or any significant transistor additions. Brown et al. [39] improve thread switching performance, one of the key problems of operating system execution, by evaluating how to include additional thread state in the processor microarchitecture. This reduces the need to go to main memory to load and store architectural state, which enables fast thread scheduling and switching both on single processors but also on multiple processors. This is similar to the approach that many-threaded GPUs have taken in recent years using hardware based thread scheduling for local decisions and only relying on software scheduling for coarse granularity load balancing.

3.2.2 Offloading for Power Efficiency

Mogul et al. [40] have 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. Mogul et al. utilize hand instrumentation of long running system calls such that if a system call is predicted to execute for a long duration, execution is migrated via the native OS process migration methodology to the lower power OS processor. While system calls are executing on the OS processor, the aggressively designed user processor can enter a low power state or be context switched to an alternative thread of execution. When the OS routine has completed execution, the user processor is powered back up,

and execution returns to the high performance user processor while the OS processor enters low power state. This migration of OS execution to a more energy efficient processor results in a net improvement in the overall efficiency of the compute system as a whole.

Process migration of this nature, while conceptually simple, is actually a very heavyweight operation, taking several microseconds to occur between cores that even share a last level cache such as the Intel i7 Sandybridge processor series. This heavyweight operation limits the opportunities for migration, and any improvement to scheduling or migration efficiency would certainly improve the overall performance. In this work they also looked at binding high rate interrupts to low power-cores, which provides a static cache partitioning manually. While this is useful from a partitioning standpoint, we will show in the final chapter of this dissertation that moving interrupts away from the cores that will ultimately handle the data results in additional cache coherence operations as the data must be accessed remotely. This approach is good for OS scalability, but bad for per operation latency.

The approach of moving applications to power efficient cores has recently come to the forefront of the industry in the mobile space where ARM has announced a *BIG.little* many core arrangement where asymmetric cores can be scheduled appropriately to match power consumption with computational efficiency. In the current Linux and ARM v7 ISA implementation these cores are not actually available to the operating system to perform task scheduling as proposed by Mogul et al. Instead the opportunity for power savings comes from shutting down the complex out of order pipeline and moving to a simple in order pipeline without any OS knowledge of this occurring. This improves the switching overhead substantially, but requires a unified register, cache, and TLB arrangement since these structures would require OS intervention to manage on process migration. This approach slightly hurts overall performance but is a win for energy savings [41], which is the driving force in the mobile market today. Interestingly, the trigger for the microarchitectural migration between the big and little cores is actually driven by the operating system without its knowledge. The migration is controlled by the device voltage and frequency scaling (DVFS) interface exposed to the OS, which scales back frequency and voltage when

it believes the core is underutilized. When the microarchitecture receives the DVFS adjustment via ACPI, it actually changes pipelines rather than performing the DVFS operation. In many ways this may be more akin to architectural reconfiguration than migration.

3.2.3 Reconfiguration for Power Efficiency

Li et al. [37] take a slightly different approach to OS execution than other proposals. They propose that rather than implementing a secondary processor, existing uniprocessors 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 offloading identification. We believe our hardware based decision engine could be utilized effectively by Li et al. to improve the impact of their optimization by allowing fine grained control over when reconfiguration should occur.

Sondag et al. [42] propose that rather than reconfiguring the processor for a single generated code, compilers should be emitting multiple versions of code that are functionally equivalent. These code versions can then be matched to available resources during phase-based tuning to maximize the power efficiency or performance of the system. They propose that producing a fat binary optimized for multiple architectures is no longer an issue with large main memory systems since the majority of the code segment will not be executed; thus there is no impact on processor caches, leading to more performance portable code regardless of the processor type it ends up running on.

3.3 Cache Partitioning

Cache partitioning is not a new idea. Stone et al. [43] look at static partitioning schemes in a uniprocessor where multiple-threads are competing for a single resource and determine that LRU is close to optimal in steady state execution, but convergence on that steady state can be slow. In their work they use offline profiling to drive their

partitioning schemes, which is impossible to do in practice. The effects of multiple processes context switching in a cache is also analyzed by Lui et al., but their analysis focuses on prefetching [44]. Operating system execution does not always occur in a different address space than the user process; thus context switching is a different problem than we are addressing here.

Su et al. [45] first examined dynamic partitioning of a shared cache using information about the recency of hits to lines in the cache to determine the utility of allocating cache resources between processes. Their way-partitioning uses additional tag information to determine the relationship between processors/threads and partitions. This is most similar to our way-partitioning scheme, except that we do not maintain per set partitioning information due to the storage overhead and dynamic search space complexities it produces.

Qureshi et al. [46] improved on the work of Su et al. with a monitoring proposal in which partition utility can be calculated and predicted on-line regardless of the application mix. It moves the monitoring away from tag based record keeping to a per-processor monitoring scheme that requires a fixed overhead per processor. This on-line monitoring and prediction allows them to update partition sets as workload mixes and requirements change, thus actively helping to optimize overall throughput of the system.

3.4 Adaptive Insert

An alternative to static or dynamic partitioning is to maintain LRU replacement but insert newly fetched lines into non-MRU positions. This has been examined for single threaded applications with insertion into, random, LRU, or adaptive locations in the LRU queue by several groups [47, 48]. It has recently been applied to shared caches as a low overhead alternative design to cache partitioning as well [49]. Cache capacity thrashing can be minimized using adaptive insert without placing restrictions on the associativity or capacity accessible by the competing threads.

Most of this work focuses on solving a specific problem where cache capacity is not sufficient to achieve line reuse by the application(s). This results in cache thrashing, as MRU inserts filter through the LRU queue, pushing out cache blocks that are going

to be reused but are beyond the reuse distance of the cache. For situations where this behavior can be identified, adaptive insert is extremely effective. The type of N+1 capacity thrashing that is handled well by adaptive insert appears to be different than the cyclical execution thrashing exhibited by OS/User Virtual threads.

3.5 Other Work

Kumar et al. [15] have shown that it is possible to implement multiple heterogeneous cores sharing a single ISA on one die. This work shows that it is possible to dynamically migrate a process to the core on which it achieves the best power/performance trade-off. While their work focuses on power savings, an average 39% energy reduction with 3% performance degradation, this work could be extended to heterogeneous cores trying to achieve improved performance. They do not examine operating system effects nor look at the operating system beyond utilizing improved scheduling ability to migrate processes to the appropriate core.

Moseley et al. [50] show that on the Pentium 4 the performance counter implementation is rich enough that dynamic scheduling of threads utilizing a performance feedback loop can result in improved processor utilization. Moseley et al. do not attempt to schedule to heterogeneous cores but show that thread migration across processors can possibly overcome the performance penalty inherent with halting and resuming execution of a context on a different processor. This work does not examine the costs of context migration and uses a much larger grain size for process scheduling than this dissertation entails. Their results however provide insight into the feasibility of process migration and control via the operating system scheduler as to the benefits of coscheduling or put otherwise removing destructive interference caused by additional processes, such as the operating system.

Redstone et al. [16] examine the percentage of time spent within the operating system for several commercial applications. They find that it is a nontrivial amount in all cases and can consume up to 70% of the total instructions executed by an application such as a web-server. This work is a major contribution to the field in that it is the first to substantially show that for an important class of applications, architectural simulations that do not take into account operating system effects

are going to be quite inaccurate. This work does not examine in great detail the breakdown of operating system execution, but provides a stepping stone for this dissertation into the types of applications for which the operating system performance can have a large impact.

Li et al. [32] examine the effect that the operating system has on branch prediction accuracy when trying to maximize performance of a single application. They find that making the branch prediction tables *OS-aware* can significantly increase prediction accuracy of most popular branch prediction schemes. Decreasing branch prediction misses can significantly speed up the performance microprocessor and increase processor utilization, thus decreasing overall energy consumption/ work unit. This work provides solid evidence that the operating system causes destructive interference with an application when allowed to execute simultaneously on a single core. We propose that removing the operating system interference, rather than adding additional hardware to existing structures, has the potential to increase performance in a similar fashion.

Hankins et al. [51] recently examine the potential of a heterogeneous multicore processor in which some of the application processors do not support full operating system functions. While not the focus of this work, it implements a system in which application code is migrated to a separate operating system processor on every system call. They implement this system in a Pentium Xeon processors microcode to catch privilege level switches to trigger their migration. Unfortunately they do not attempt to detail the migration overhead or efficiency of executing code on behalf of the application. We believe that our proposal and close examination of their approach will show that their migration strategy, while feasible, is far from the most efficient. We propose that a coarser grained migration strategy is required to overcome the performance penalty of thread migration.

Davis [52] describe a custom built asynchronous architecture in which two cores operating as symmetric multiprocessors actually achieve improved performance by statically partitioning the operating system on to one core and the application code onto the other. This early work exposed the significance of the destructive interference the operating system can have on application performance. This work inspired some

of the initial results for this proposal with regards to partitioning interrupt handling and system daemon execution to single processors in a SMP configuration.

There is a larger group of papers [22–24, 53] that categorize some of the performance gaps we are seeing in current operating system execution. These papers often describe the problems in depth but fail to propose solutions that are beyond a rewrite of that specific operating system function. Very few examine the potential for architectural support for these features. Generally speaking the operating system developers tend to be consumers of architectural change, not motivators. We believe there is substantial gain to be made by including them in the design process. Mogul et al. [54] have provided some general guidelines to processor architects they might use for feature design to ensure that OS architects can make use of those features in new and legacy operating systems alike.

The Barrelfish project [55, 56] is one project attempting to solve the OS scalability problem from the bottom up. They have taken extensive measurements of modern CMP and SMP shared memory machines to determine the cost of hardware cache coherence versus message passing with a share-nothing design goal. They have designed a complete research operating system from the ground up that shows that it is possible to achieve good latency and improved scalability beyond that of traditional operating systems such as Linux and Windows, which rely heavily on hardware cache coherence for synchronization primitives. The Barrelfish project, however, does not propose new hardware features that might improve their OS throughput; they simply demonstrate that they can achieve good performance by ignoring some of the features hardware provides that may simplify microprocessor design if they could do away with supporting.

Several groups of researchers [34, 57, 58] in operating systems have looked into architectural features that are commonly exercised by the operating system such as the virtual addressing system, and work continues today with transparent superpage support being a hot topic in the Linux kernel community with the rise of big data. Because of this, a large body of work has gone into speeding up context switching while maintaining or increasing process isolation. Maintaining per process virtual address spaces is a costly operation, which provides this mechanism in almost all

current microprocessors. Thus, hardware support for a single flat memory space in which the operating system uses an alternate process isolation scheme has been explored well. Unfortunately, legacy issues both in operating systems, as well as hardware instruction set architectures, cause these features to stall before reaching production.

Superpage support is a prime example of slow adoption due to both hardware limitations and OS implementations issues. Though superpage support has existed in x86 processors since as early as 2002, Linux only began supporting it in 2004 with special purpose file systems that had to be preallocated and memory mapped into your process address space (TLBfs). In 2008 superpages were supported via traditional malloc semantics, but as of this dissertation they still have to be preallocated at boot time and are not supported for DMA, swapping, or other common operations. In addition to the slow OS support around superpages, even when they are in use, most processors implement an extremely low, four in many cases, number of TLB entries for superpages. This means the TLB coverage for 2MB superpages is actually worse than for traditional 4k pages, defeating one of the major features of using superpages in the first place.

Similarly, another category of operating system research looks into how the operating system can, and should, abstract interfaces to devices to allow efficient access from user code to physical devices. Less than 10 years ago, any device access typically required a full address space switch for the application to read or write memory to an I/O device. The addition of DMA and zero copy protocols alleviated this problem to a large extent, but it has not settled the debate on the proper design model. Engler et al. [59, 60] argue that abstraction of device interfaces by the operating system inherently cause undue performance penalties and that architecture support for multiplexed devices should exist at the hardware level. Even the notion of allowing devices to use interrupts to break execution on the processor has been called into question because when the interrupt comes in the OS has no idea how critical this particular interrupt actually is. Thus all interrupts tend to be serviced with high priority, when we would much prefer to defer some interrupts at times, providing essentially interrupt scheduling [61].

One area where OS support and hardware have worked well together and moved quickly is in the area of virtualization where the performance of full virtualization on top of the existing OS was so poor it made virtualization unusable in many use cases. On the operating system side, the Xen and KVM virtualization projects were quick to adopt paravirtualized drivers that reduced the overhead of effectively running two OS stacks on top of a single device to multiplex on it. Nested pagetable support, such as AMD's AMD-V, improves memory management by allowing a single page table walk when translating from guest virtual memory into physical memory directly rather than adding another virtualized page mapping layer. While these technologies have rapidly been adopted because the cost of further virtualization was prohibitive to performance, fewer attempts have been made at improving single OS performance on native hardware. High performance networking and storage devices have working bodies trying to drive access partitioning further into the hardware with technologies like SR-IOV where devices can directly take virtual addresses rather than physical and rely on a hardware IOMMU to do the translation for them to the physical BAR required for DMA. Paravirtualization, nested pagetables, and SR-IOV and I/OAT are all targeting improving aggregate throughput of a multiplexed system; however, they do not improve single threaded performance and typically come at a detriment to single IO latency, which we will explore more in the last chapter of this dissertation. In addition to these industry efforts, there are academic efforts [62–64] exploring what the right tradeoff is in systems between having centralized control (CPUs) versus peripherals (NICs, storage). One of the key drivers is datacenter power where extraneous transfers between peripherals and system memory can result in high power usage with no performance benefit. Ram et al. [65] attempt to blend the lines between a switch and a NIC in a virtualized environment by striking the appropriate balance of hardware and software driven features and find that performing the *last hop* within the hypervisor is unlikely to scale up. By implementing a switching accelerator directly on hardware to control the last hop routing to the virtual machines, they are able to outperform the software only switching implementation in the Xen hypervisor.

Condit et al. [66] have examined how to remove filesystem overheads for NTFS in preparation for upcoming byte addressable persistent memories such as PCM with

a filsystem they call BPFS. They show that when operating against a RAM disk there is room for nearly 100% performance improvement compared to filesystem implementations optimized for disks.

Finally, there is an active body of research [17, 19, 21, 67, 68] into microprocessor simulation that is critical to any architecture research. Simulator speed and accuracy of simulation are typically negatively correlated. As such, extremely accurate simulations can take longer than is practical when achieving research. Because of this, often simulation details are glossed over, in particular operating system execution. Implementing accurate simulation of OS execution is a significant undertaking because it requires writing models for I/O devices and other structures a typical processor simulation does not support. Fortunately Simics, ASim, M5, and others are starting to support operating system and I/O making the work in this proposal feasible. The RAMP project [69] has proposed that architectural level simulation could be sped up by using FPGAs by several orders of magnitude, but the toolchains are a research vehicle and not ready for production use as a simulation environment.

CHAPTER 4

OS OFFLOADING TO APPLICATION SPECIFIC CORES

In the era of plentiful transistor budgets, it is expected that processors will accommodate tens to hundreds of processing cores. With core counts escalating and no clear solution regarding their efficient utilization, we can consider dedicating some on-chip cores to common applications. In a homogeneous chip multiprocessor (CMP), dedicating some fraction of the total cores to a specific application can yield improvements because isolation can lead to noninterference in cache, CPU, and branch predictors. In a heterogeneous CMP, customization of these cores could allow applications to execute faster or more power efficiently. There are numerous suggestions that some workloads are well positioned to leverage heterogeneous chip multiprocessor platforms [15, 36, 37, 40, 55, 70–72]. Recently ARM has released an architecture they term "BIG.little," which is a heterogeneous mobile processor architecture in which there are paired high performance and low performance cores and execution can be migrated between them to save power. The power savings can be as high as 17% according to their marketing material, but comes at a 5% performance penalty. This architecture has heterogeneous pipelines but actually shares the same caching and TLB structures because specialization of caches would consume too many transistors on these small mobile chips.

One common application that is poised to benefit from both isolation and customization is the operating system: it executes on almost every processor and is invoked frequently – either by applications (to perform privileged operations on their behalf) or by periodic interrupts (to perform system-level management and book-keeping). There is strong evidence [16, 22, 33, 35–37] that the past decade of

microarchitectural research has had little impact on improving the performance of the OS. This can be attributed to many factors: OS invocations are typically short, have hard-to-predict branches, contain little instruction-level parallelism (ILP), and suffer from cache interference or cold cache effects. As a result, OS code executes on overprovisioned hardware (for example, floating-point units, a large reorder buffer, and large issue width). These structures are highly inefficient in terms of power consumption and are not utilized effectively during OS execution.

Studies [16, 36, 38] have shown that operating system code can constitute a dominant portion of many important workloads such as webservers, databases, and middleware systems. 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. Past work [36, 38, 40] has demonstrated the potential of core isolation and/or customization within multicores to improve OS efficiency. The motivations for improving OS execution range from performance optimization [38] to improved power efficiency [40]. In these works, the fundamental mechanism for OS isolation, *offloading*, remains the same. Offloading implementations have been proposed that range from using the OS' internal process migration methods [40] to layering a lightweight virtual machine under the OS to transparently migrate processes [38]. In all the studies we are aware of the decision process of which OS sequences to offload has been made in software, utilizing either static offline profiling or developer intuition. A related alternative to offloading is the dynamic adaptation of a single processor's resources when executing OS sequences [37, 73]. While the dynamic adaptation schemes do not rely on offloading, the decision making process on when to adapt resources is very similar to that of making offloading decisions. Even though all of these static decision making processes (based on profiling with software instrumention) result in lower hardware overheads, they lose flexibility of temporal changes in adaptation requirements and the accuracy afforded by an on-line, hardware based scheme.

In this dissertation we analyze operating system offloading (for performance) as a case study of the design factors that influence offloading success. We study the trade-offs in offloading OS execution to a specialized core *independent* of the offloading

mechanism. Being implementation independent allows us to vary parameters such as migration overhead latency, offloaded OS routines, and the constant overhead of offload decision making. We analyze the design space that encapsulates existing proposals and future implementations to understand how these design parameters influence the performance of the OS offloading solution.

Our eventual goal is to determine *during workload execution,* which OS sequences to offload that maximize its performance. We first examine the overhead of software based offloading decisions and find that this approach can contribute nearly zero or as much as 94% overhead for short OS routines. We propose a hardware operating system run-length predictor that requires less than 2KB of storage overhead and returns predictions within 1 cycle at a target frequency of 3.5Ghz. This predictor provides an exact prediction of OS-Runlength 71.2% of the time. The prediction is within ±5% of actual OS run-length 92.3% of the time.

Across different workloads, it is unlikely that a static implementation of OS offloading will result in maximal performance for all workloads. To overcome workload induced variation, we propose a feedback directed offloading mechanism that can dynamically adapt the amount of offloading that occurs at runtime. We show that incorrect decisions about which OS sequences to offload can result in up to 42% variance in application throughput. Utilizing OS run-length predictions and performance feedback to adjust an offloading threshold, offloading decisions can be made in just one cycle while still allowing dynamic adaptation to workload variance. The result of this study is a general purpose decision mechanism that can be applied to almost all existing OS offloading proposals as well as dynamic processor reconfiguration. We believe this mechanism can improve the performance of all existing proposals while simultaneously lowering their implementation complexity.

To evaluate this work we use the Simics simulator executing a full Linux distribution based on kernel 2.6.28 executing on a hypothetical architecture implementing the Sparc V9 ISA. The Linux distribution was pared down by hand so that additional services and daemons were not executing that were not critical to application execution such as sound cards, additional hard drives, USB, etc. The processor modeled was based on the ultraSparc III architecture, which contained a 32KB i-cache, 64KB

d-cache, and 1MB unified L2 cache. We modeled a 1Ghz frequency and used a fixed memory latency of 100ns to speed up simulation time.

4.1 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 4.1, 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 that 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 4.2 for a distribution of the length of the privileged instruction runs in terms of number of instructions) and have been removed from all further results to not bias them compared to other architectures such at ARM or x86.

4.2 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 wildly different depending on many factors, such as system call use by applications, kernel housekeeping, and device interrupts. Figure 4.2 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 nontrivial

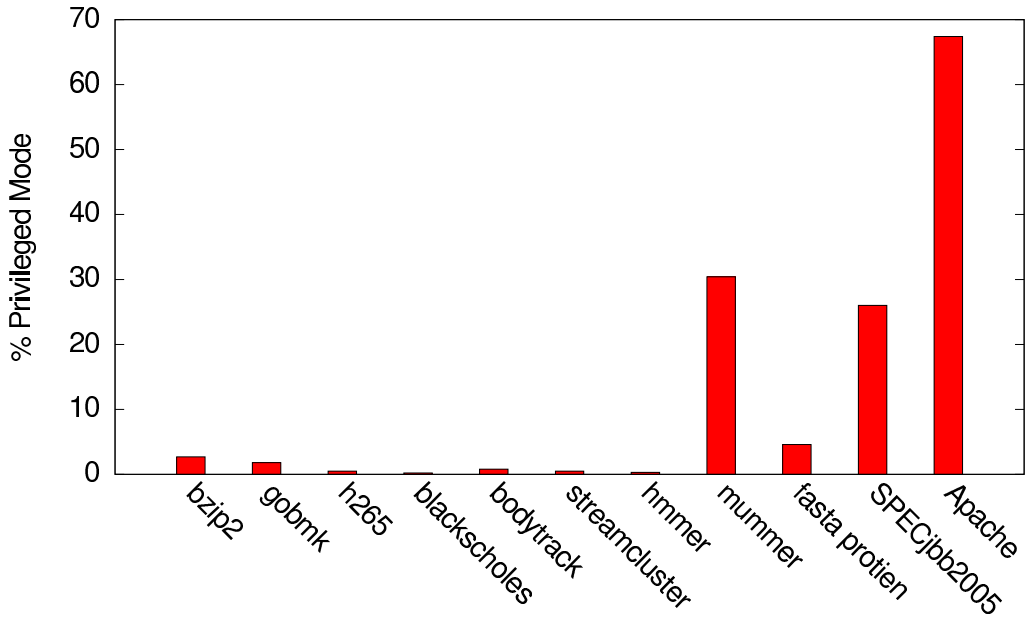

Figure 4.1. Percentage of workload instructions that occur within operating system code (privileged mode).

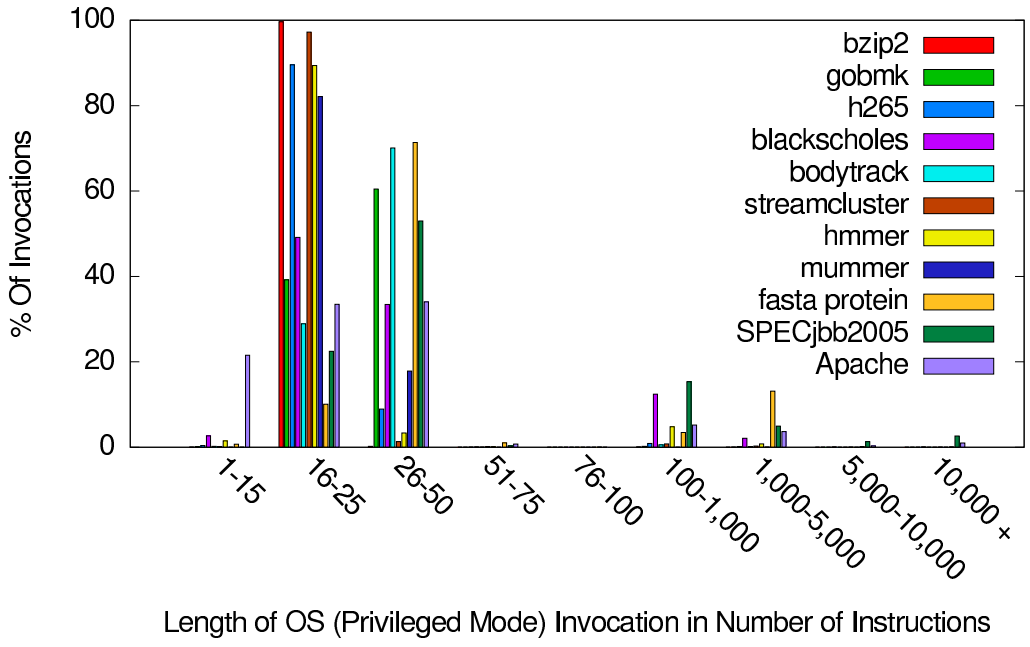

Figure 4.2. Length of OS system calls and other OS execution during benchmark execution.

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 dissertation chapter.

4.3 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. Figures 4.3, 4.4, and 4.5 show the causes of OS induced eviction at the L1 and L2 levels. Apache and SPECjbb2005 attribute between 25–30% 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.

Offloading execution from a traditional general purpose microprocessor is not a new idea. In the past, either due to legacy compatibility and/or processor design constraints, it might not have been possible (or more efficient) to implement a piece of hardware only to improve offloading performance. However, today transistors have become so abundant that offloading can be done as a design optimization. For instance, in the 1980s floating point hardware often existed as a coprocessor that could be plugged into an additional socket on many motherboards. This essentially amounted to offloading floating-point execution. Until recently, the memory controller for the DRAM subsystem was implemented in the north-bridge, but recent CPUs now have an integrated memory controller. This trend is pervasive throughout a computer

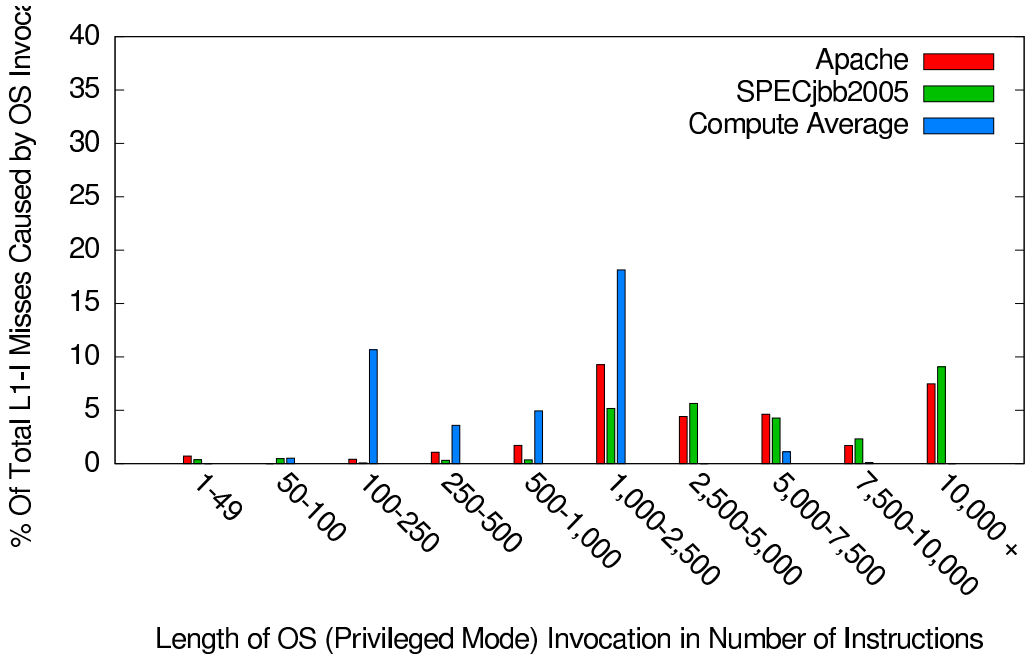

Figure 4.3. Percentage of L1 instruction cache evictions caused by the operating system, broken down by OS execution length.

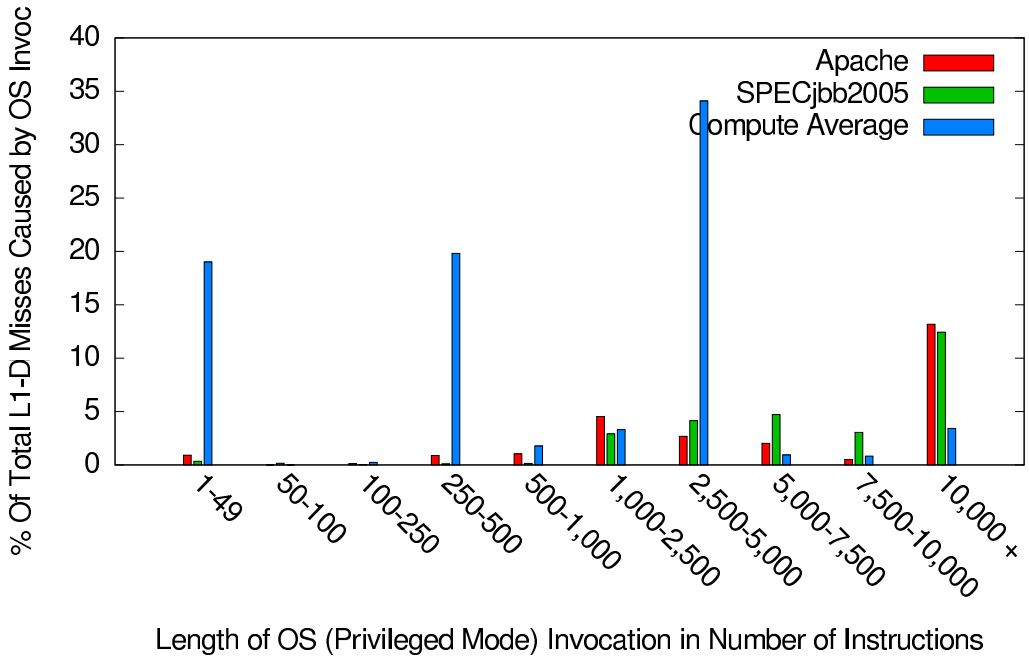

Figure 4.4. Percentage of L1 data cache evictions caused by the operating system, broken down by OS execution length.

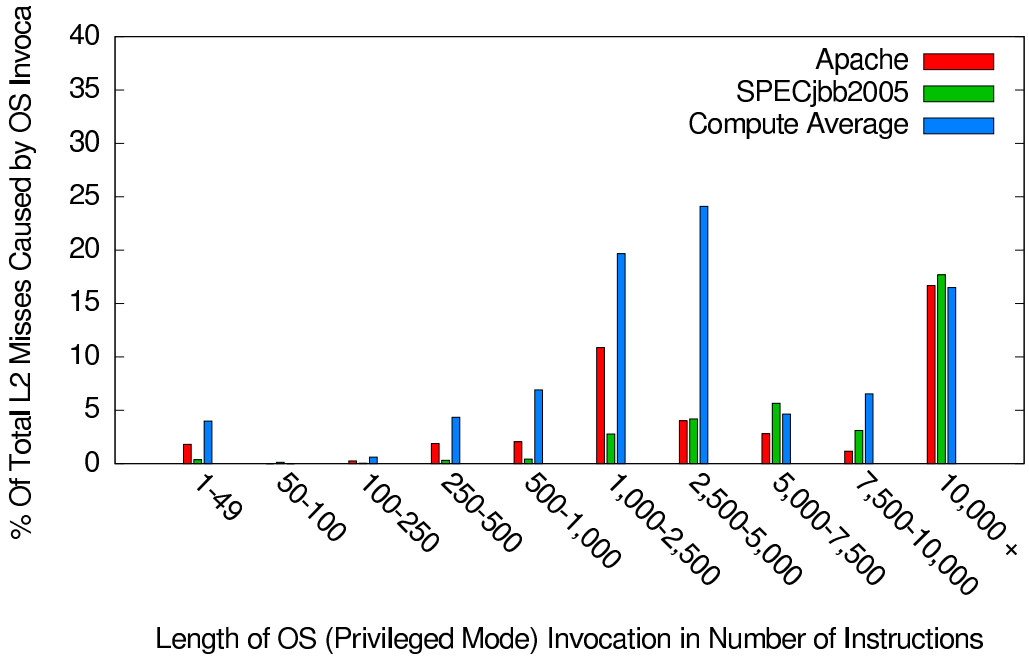

Figure 4.5. Percentage of L2 (shared) cache evictions caused by the operating system, broken down by OS execution length.

system's design: 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. Through time, computer architecture has evolved and incarnations of offloading have been proposed, evolved, failed, and been revisited. Only those proposals that achieve the right balance of performance, opportunity cost, and design complexity endure.

Our goal in this work is to propose mechanisms to improve overall system performance by offloading OS execution. We believe due to large transistor budgets and recent advances in CMPs, OS offloading provides an opportunity to improve performance and reduce power consumption. It is, however, critical to note that workload behavior plays an important role here. Figure 4.1 shows the percentage of total instructions that occur within the operating system for a variety of workloads. While traditional high-performance computing (HPC) workloads like those found in the PARSEC [74], SPEC-CPU [75], and BioBench [76] suites exercise the operating system very little, a class of server workloads executes a significant, and sometimes

dominant, portion of their total instructions within the operating system. It has been shown that for such server workloads, OS performance can be the limiting factor in overall application throughput [16].

In the future, it is possible that much of the world's computing energy budget will be dominated by datacenters and cloud computing infrastructure. In such computing platforms, many different virtual machines (VMs) and tasks will likely be consolidated on many-core processors [77]. 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. Further, it is possible that many of these VMs will execute similar codes or share the same data structures – this suggests that VM or OS invocations from each application be off-loaded to a few cores that are best suited for their execution and already have warmed up caches. Clearly, the off-loading of OS execution can be vital in such systems. While this paper tackles a simplified view of the problem, we believe that the resulting insight will play a strong role in determining the design of future many-core processors that will, to a large extent, be housed in server racks.

4.4 Common Characteristics of Offloading Implementations

While offloading and reconfiguration proposals have different goals, we can define four distinct phases that are common to any OS offloading implementation.

- **Decision Cost** is the fixed overhead that must be incurred to make a decision if offloading should take place. For all the proposals described above, researchers propose instrumenting system calls that are expected to exceed some threshold of execution length. This instrumentation is executed for every system call regardless of the offloading decision. It is advantageous to be able to offload as many OS sequences as possible; uninstrumented routines are not possible candidates for offloading. Instrumentation overhead is insignificant in long running system calls; however, Figure 4.6 shows that a significant portion of the total invocations are very short in duration. As we will show in Section 4.6.1, even minor software instrumentation (especially for system calls with relatively

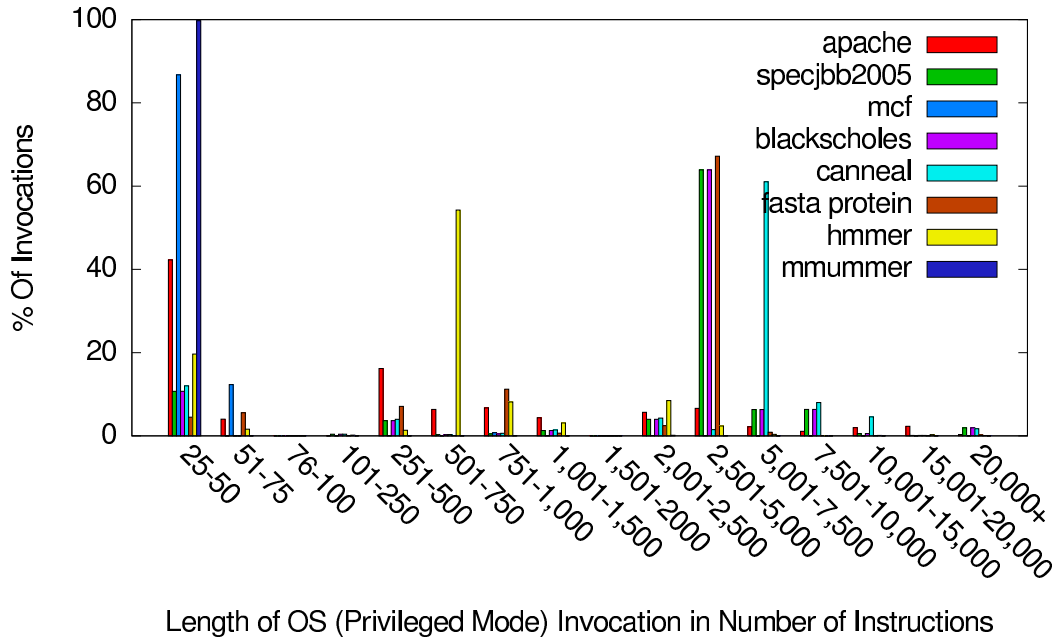

Figure 4.6. Percentage of OS invocations broken down by execution length (excluding short sequences particular to SPARC ISA).

short duration) is detrimental in terms of performance, and unmaintainable due to the large number of system calls present in modern systems.

- **Decision Making** is the process during which various parameters (like system call input arguments etc.) are evaluated to determine if offloading should occur, or, if execution should remain on currently executing core. Execution on modern microprocessors is not deterministic due to external interrupts and devices, thus any decision has to be made probabilistically. For example, at decision time we might predict that a syscall may require only 50 instructions to execute, but during execution this syscall may be preempted by a 10,000 instruction long OS sequence initiated by a device interrupt. Furthermore, it is also possible that even for a single system call, the objective function (performance or energy efficiency) is maximized by selectively offloading this call. The *read* system call exemplifies this; *read(N)* attempts to read N bytes from a file descriptor. The duration of this system call can vary by orders of magnitude if *read* is called to bring in 1 byte or 10KB from disk. *Decision making* must take into account this variability in execution length, further complicating the policy decisions to

be implemented.

- **Offloading** occurs *only* if the decision making portion of the offloading scheme results in a positive result. If so, the migration implementation is invoked and execution is passed from the user processor to the OS processor. When OS execution has completed, the migration implementation is invoked a second time, but in reverse, and execution is restored on the user processor. The latency and operations required to implement offloading are fixed by the migration implementation, taking into account both hardware and software capabilities. In this study we do not compare migration implementations but instead recognize that different implementations exist. We perform a design space study with varying penalties for migration, without being tied to any one implementation.
- **Feedback** optionally occurs after an offloading sequence was executed. Because of the nondeterministic nature of execution on modern processors and nondeterministic execution of OS code itself, it is useful to provide performance based feedback to the *Decision Making* process. Without feedback, variation in application workload, hardware parameters, and migration implementation reduce the maximum performance gains achievable by an offloading implementation. For example, choosing to offload the *read* system call for one workload may result in a net speedup, but for another workload it may not be beneficial. This can only be determined empirically at run time by utilizing performance counters and implementing a feedback loop that can influence *Decision Making* policies.

4.5 Experimental Methodology

In this work we examine application performance as a case study of operating system offloading. Prior work has shown [38] that offloading OS system-calls to remote processors has the potential to speed up application throughput. To examine the design space of OS offloading we use a cycle-accurate, execution-driven simulation to model full OS execution during workload execution. We then parameterize the *migration implementation* so that we can examine the effects of varying latency implementations on the offloading decision policy. Because this is a design space

exploration and we parameterize the cost of migration implementation, one should not view our performance results in isolation. In many cases we are doubling the number of transistors by implementing a second core with a second cache so performance should be expected to improve. Instead our results should be viewed relative to other points within the design space for which a real solution can be implemented. It is trends within this design space that we are attempting to illuminate.

4.5.1 Processor Model

We use Simics 3.0 to drive our execution based simulations based on an in-order UltraSPARC core. 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 multithreading [16] (for example, the Sun Niagara and Rock designs and the recent Intel Atom design).

On the SPARC platform, determining if OS code is executing can be figured out by examining the PSTATE register [78]. The PSTATE register 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 this register to determine what is running as part of the OS code and what is running as the User code by observing the execution mode bit. Via this definition, system calls that occur in privileged mode but within the user address space are captured as part of our OS behavior as well as 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 offloading 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 of the 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.

4.5.2 Caches

For all our simulations, Table 4.1 shows the baseline cache parameters. Modifications to this cache are described in the relevant sections. Our timing parameters were obtained from CACTI 6.0 [79] targeting a frequency of 3.5 GHz. At this frequency an optimistic memory latency of 350 cycles is used in all experiments (based on real machine timings from Brown and Tullsen [39]). In the case of offloading, we implement 2 processors with private L2s that 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 multicore system). We model different L2 sizes (either 512KB or 1MB) to ensure an apples-to-apples comparison for each experiment (this is specified while describing the experiment).

4.5.3 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 [74], BioBench [76], and SPEC-CPU-2006 [75] 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 and SPECjbb2005 comprise our server oriented workloads. All benchmarks utilize a 1:1 mapping between cores/threads except Apache, which self tunes thread counts to optimize throughput.

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.

Table 4.1. Baseline cache configuration.

L1 Inst.	32KB, 64B line, 2-way LRU, 1 cycle
L1 Data	32KB, 64B line, 2-way LRU, 1 cycle
L2	1MB, 64B line, 2-way Banked, 16-way LRU, 12 cycle
Memory	350 cycle uniform latency

4.6 Minimizing Opportunity Cost

4.6.1 Overheads of Migration

Thread migration to an OS core minimally requires 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 are data in cache on the user processor that must be accessed by the OS core, they must be transferred to the OS core (automatically handled by coherence mechanism). Typically, cache data are not aggressively prefetched into the OS core to avoid pollution and wastage; instead they are fetched on a demand basis, leading to longer latencies per access until the cache is warm. To our knowledge, approx. 6,000 cycle round trip time is the fastest and fully implemented thread migration implementation [80] available.

Recent work suggests that hardware support for book-keeping and thread scheduling (normally done in software by an OS or virtual machine) can lower the basic execution migration cost to as few as several hundred cycles for SMT capable processors. This comes at the expense of an additional processor to compute and maintain this state. Our study evaluates several optimistic design points for offloading overheads in anticipation that a low overhead solution, such as that of Brown et al. [39], could be applied to privilege mode execution.

Migration overheads are also impacted by the specific implementation that determines whether to offload or not. Previous proposals have involved a VMM to trap OS execution and follow a static policy based on off-line profiling [38] or have hand instrumented the operating system within various routines to predict in software the duration of execution [32, 40]. These software-based mechanisms may incur hundreds

of cycles of overhead, whereas our predictor-based mechanisms rely on hardware support that can accomplish the decision-making in a single cycle. Even a few hundred cycles may be dwarfed by the long duration of some offloads, but Figure 4.6 shows that to minimize interference we must consider substantially shorter OS sequences. Instrumentation costs are incurred every time an offloading decision is *considered*; thus for workloads where OS activity is frequent, it is *critical* to minimize this cost. Additionally this instrumentation must occur in all system calls considered for offloading, a daunting task considering the number of system calls in modern operating systems shown in Table 4.2.

To illustrate this point, consider a software based scheme where OS system calls are instrumented by hand to include a branch that determines if offloading should be done based on looking up some register's contents. In Listing 4.1 we show the basic code for getpid syscall implementation in OpenSolaris. Listing 4.2 shows how this code needs to be instrumented to support software-based offloading. Note that the instrumented syscall first needs to read a register value to determine if the current call is invoked with parameter above a threshold so that it is useful to offload it.

Such instrumentation has to be done by hand for all frequently executed system calls, and furthermore, the threshold value either needs to be a static value for each kind of syscall, or if it is read from some architected state, then there needs to be more code to read that value. Both these options increase the overhead (in terms of new instructions) of the system call itself and increase the opportunity cost for offloading. To measure this increased overhead, we compared the assembly instructions the above two listings compile to (for SPARC ISA).

We found that adding a simple offloading branch that determines offloading based on a static threshold increases the assembly instruction count from 17 to 33 for this trivial example. Note that this increase in instructions is without counting the instructions for reading the register value (*read_some_reg_val*() function). Considering that there are numerous short syscalls like getpid, the overhead induced by these new instructions can be quite large (>90% for this example). Note that this overhead is incurred on every syscall invocation, regardless of whether the syscall is off-loaded or not.

Table 4.2. 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

Considering Figure 4.6, which shows that there are a large number of invocations that are shorter than 100 instructions, adding even short code to handle offloading can hurt performance substantially. In the following sections, we propose hardware based schemes, which are free not only from having to hand instrument each syscall, but also use feedback to make decisions about the appropriate threshold to offload execution. This eliminates the need to profile operating system code or manually modify hundreds of operating system routines.

4.6.2 Hardware Prediction of OS Run-Length

While offloading may occur for differing reasons, the basis for making the offloading decisions is almost exclusively the estimated run-length of the OS routine. This is so because OS run-length is a single well defined parameter on which decisions can be based. This is also backed up by intuition. For example, given an opportunity cost of 100 cycles and a migration implementation of 5,000 cycles in each direction it is extremely unlikely that an OS sequence merely 10 cycles in duration could benefit from offloading. An OS sequence 20,000 cycles long has a much higher probability of improvement. While run-length is not the direct objective of power or performance offloading, it is the common factor used in most offloading policies, and thus we choose to use it here as our metric of interest.

Operating system offload for performance should almost always happen only when executing sequences that result in a net speed-up of the total system. As seen in Section 4.6.1, system calls can have dynamic behavior based on their inputs. Hence,

Listing 4.1. Original getpid system call.

```
int64_t getpid(void)
{
  rval_t  r;
  proc_t  *p;
  p = (kthread_t*) threadp();
  r.r_val1 = p->p_pid;
  r.r_val2 = p->p_ppid;
  return (r.r_vals);
}
```

Listing 4.2. getpid instrumented for offloading.

```
some_reg_val = read_some_reg_val();
if(some_reg_val > static_threshold)
{
  offload();
}
else // don't offload
{
  p = (kthread_t*) threadp();
  r.r_val1 = p->p_pid;
  r.r_val2 = p->p_ppid;
}
```

the design of a good dynamic offload mechanism is contingent on our ability to accurately predict the length of an OS invocation. We propose a new hardware predictor of OS invocation length that XOR hashes the values of various architected registers. The intuition for this predictor is based on the fact that OS behavior is explicitly controlled by the architected state of the processor at run-time. Therefore, it should be possible to predict the routine that will be called (and thus its length in instructions) by examining the architected state of the machine.

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. 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 4.7.

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 1.6KB storage space. A direct-mapped RAM structure with 1500 entries also provides similar accuracy.

Averaged across all benchmarks, this simple predictor is able to precisely predict the run length of 71.2% of all privileged instruction invocations and predict within ±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 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 that are not part of the processor state at prediction time. These prediction inaccuracies are part of nondeterministic execution and can not be foreseen by software or other run length prediction implementations. Fortunately, these interrupts only 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 offloading possibly not occurring, based on a threshold decision. More often than not, offloaded sequences end up being of longer duration than predicted.

While the hardware predictor provides a discrete prediction of OS run-length, the switch trigger must distill this into a binary prediction indicating if the run length exceeds N instructions and if core migration should occur. Figure 4.8 shows the accuracy of binary predictions for various values of N. For example, if offloading should occur only on OS invocation run lengths greater than 500 instructions, then our

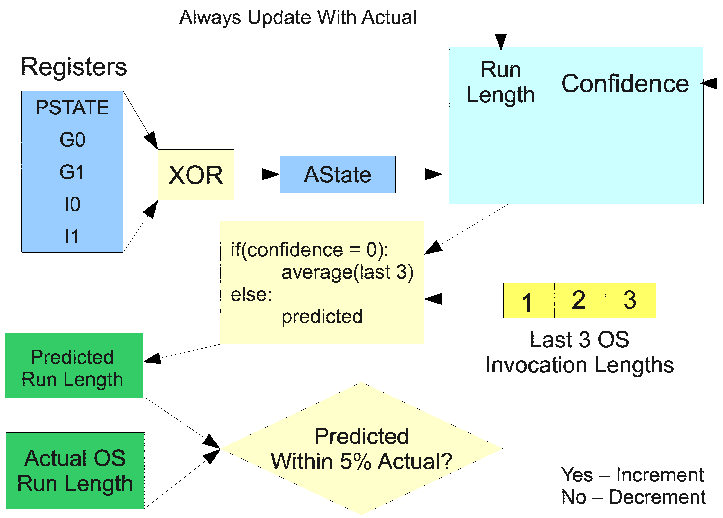

Figure 4.7. OS run-length predictor with configurable threshold.

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. While more space-efficient prediction algorithms possibly 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 in either software or hardware. The interference of these interrupts is infrequent enough, as evidenced by the >93% correct prediction rate in all cases, that we do not investigate further improvement. One technique that we could explore, however, is ignoring the prediction and feedback loop if the trap was generated externally rather than via a systemcall. While this does not solve the migration question for interrupts, it may further improve prediction accuracy for the systemcalls we are targeting.

4.6.3 Dynamic Migration Policies Based on Feedback

The second component of a hardware assisted offloading 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

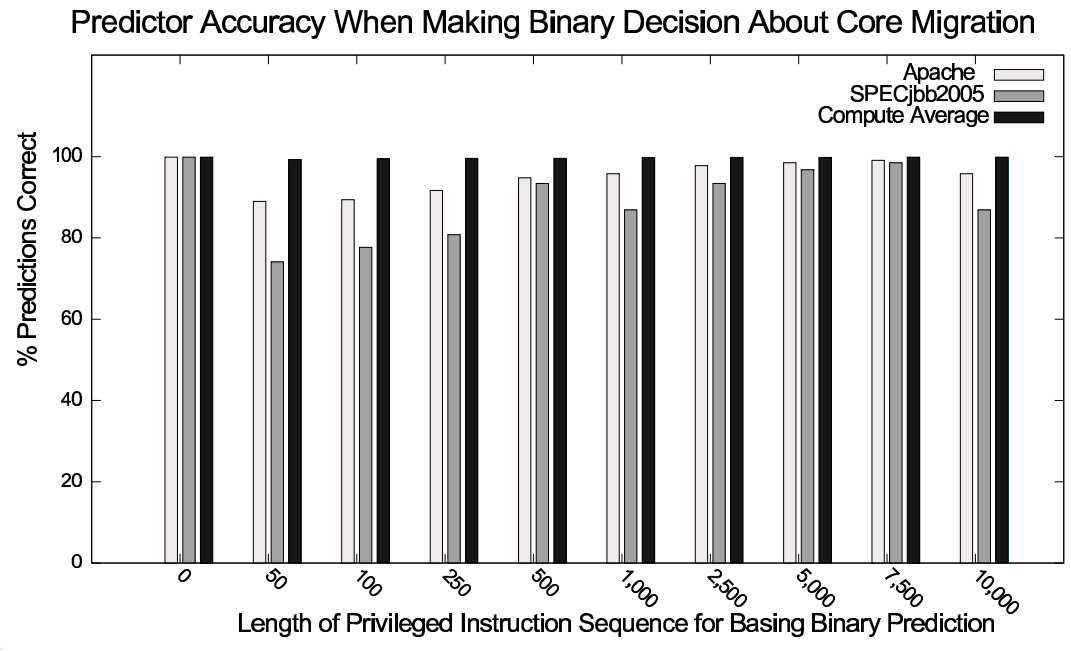

Figure 4.8. Binary prediction hit-rate for core-migration trigger thresholds.

compared to software instrumentation of system calls is minimal.

For this estimation of N, we rely on algorithms described in past work to select an optimal hardware configuration [81]. 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. 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 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 offloading 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 of 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 100M instructions. We

then again perform a 25M instruction sampling of two alternate values of N. If our threshold appears to still be optimal we then double the execution length (to 200M) instructions before sampling again to help reduce sampling overhead. If at any point our current N is found to be nonoptimal, the execution duration is reduced back to 100M instructions.

For most of our benchmark programs when looking at epochs larger than 100 million instructions, there were few phase changes that resulted in N changing throughout execution. For our experiments we use very coarse grained values of N found in Figures 4.9, 4.10, and 4.11. Increasing the resolution at which N can vary will increase the performance of the system but comes at the expense of increased sampling overhead. While there are quite a number of numeric parameters in this algorithm, we have confidence they are reasonable, though possibly not the best possible choices. We built confidence by sweeping the static parameters across a range of values that were initially picked based on intuition choosing the best. While this may have led us to a local maxima versus the global maxima, they also make sense intuitively and balance the sampling overhead versus possible increase in accuracy.

4.7 Understanding the Impact of OS Run-length on Offloading

4.7.1 Offloading for Performance

We next evaluate a predictor-directed offload mechanism. On every transition to privileged mode, the run-length predictor is looked up, and offloading occurs if the run-length is predicted to exceed N (we show results for various values of N). For this experiment, the 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 512 KB 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 the opposing core (a coherence miss). If this happens, we assume 6 cycle delay for directory look up and an additional 6 cycle delay for a cache to cache transfer.

In Figure 4.6 we saw that short OS sequences dominated the total number of OS invocations. However Figures 4.12 and 4.13 show that while long OS routines (>

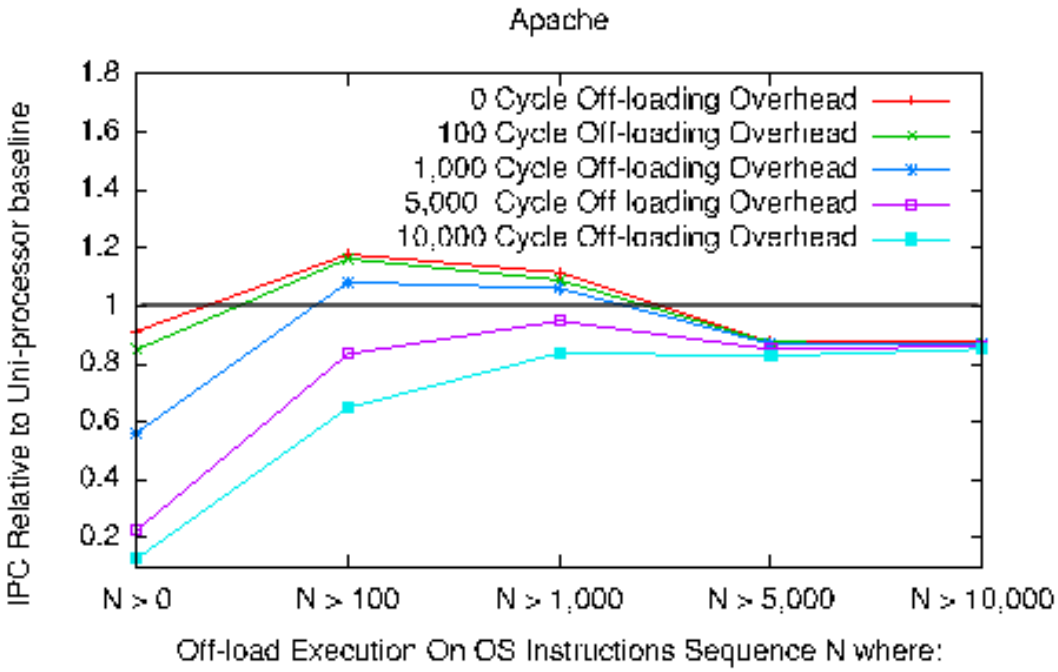

Figure 4.9. Normalized IPC relative to uniprocessor baseline when varying the offloading overhead and the switch trigger threshold for a webserver workload.

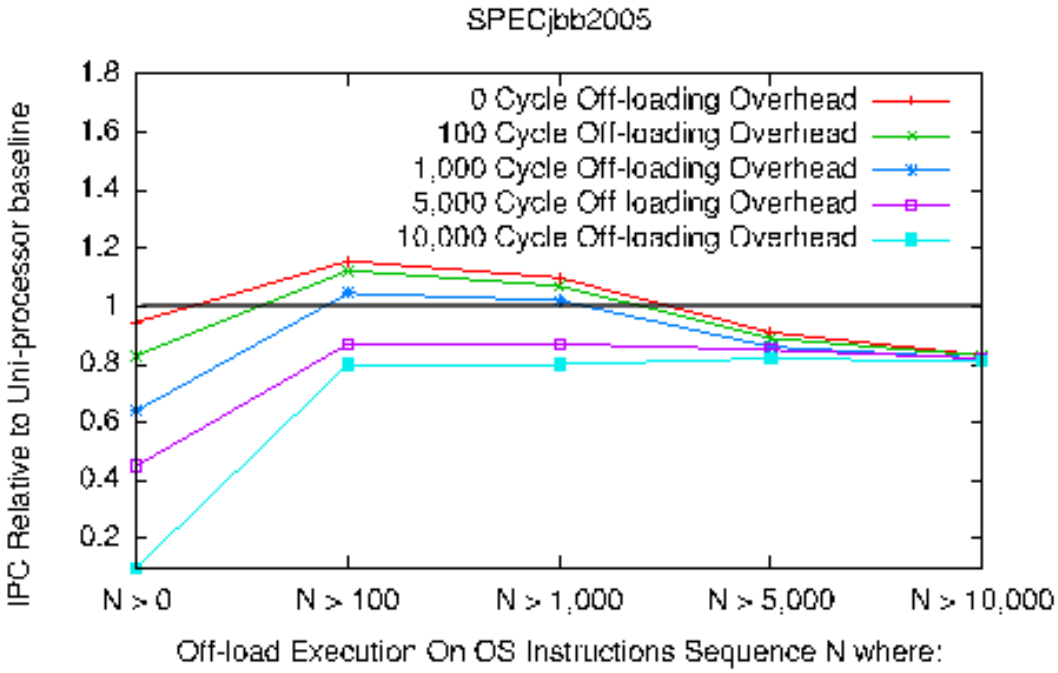

Figure 4.10. Normalized IPC relative to uniprocessor baseline when varying the offloading overhead and the switch trigger threshold for a middleware workload.

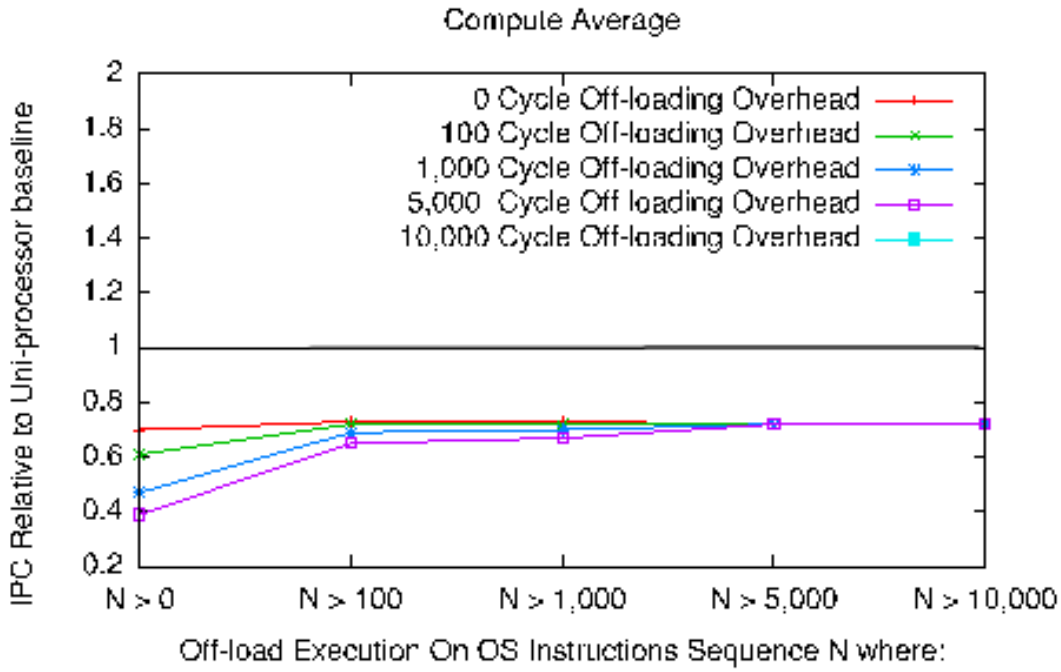

Figure 4.11. Normalized IPC relative to uniprocessor baseline when varying the offloading overhead and the switch trigger threshold for a compute bound workload.

10,000 instructions) only comprise a few percent of the total invocations, due to their long duration they dominate the total number of OS instructions executed. This leads to nontrivial design choices – it is not clear via intuition if optimal performance should be expected when offloading long and infrequent OS sequences (where migration overheads are relatively low) or short frequently executed OS sequences (that have a higher chance of reducing cache interference). To shed light on this, we examine performance as a function of the off-load threshold N.

Figures 4.9, 4.10, and 4.11 show the IPC performance through offloading, relative to a baseline that executes the program on a single core containing 1MB of L2 cache (the combined total of both cores in our offloading case) with a 12 cycle access time. A different graph is shown for Apache, SPECjbb2005, and compute-intensive programs. Each graph shows the threshold N on the X-axis and a different curve for various execution offloading delays. We evaluate multiple offloading delays because offloading implementations vary and we expect this cost to decrease in the future. Figures 4.9, 4.10, and 4.11 help identify 3 major trends about OS offloading.

- **Offloading latency is the dominant factor in realizing performance**

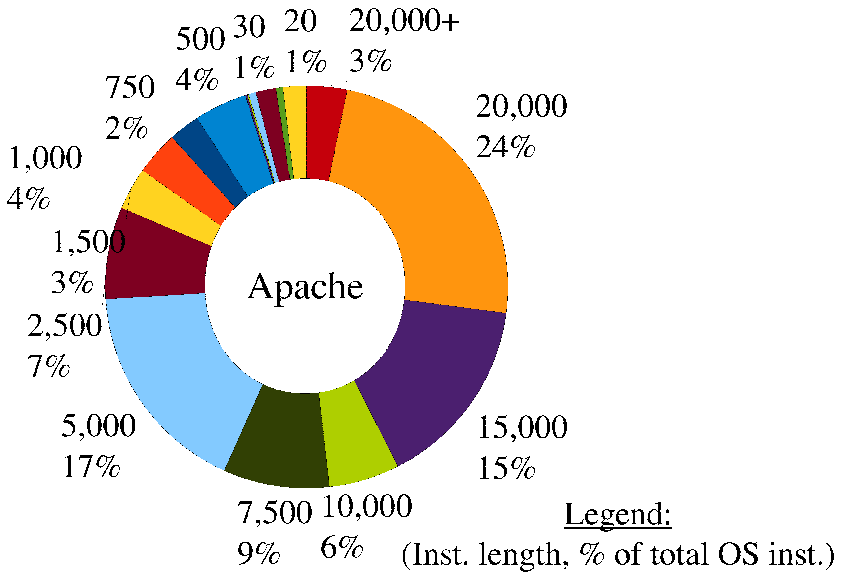

Figure 4.12. Percentage of total OS instructions by invocation instruction-length for webserver workload.

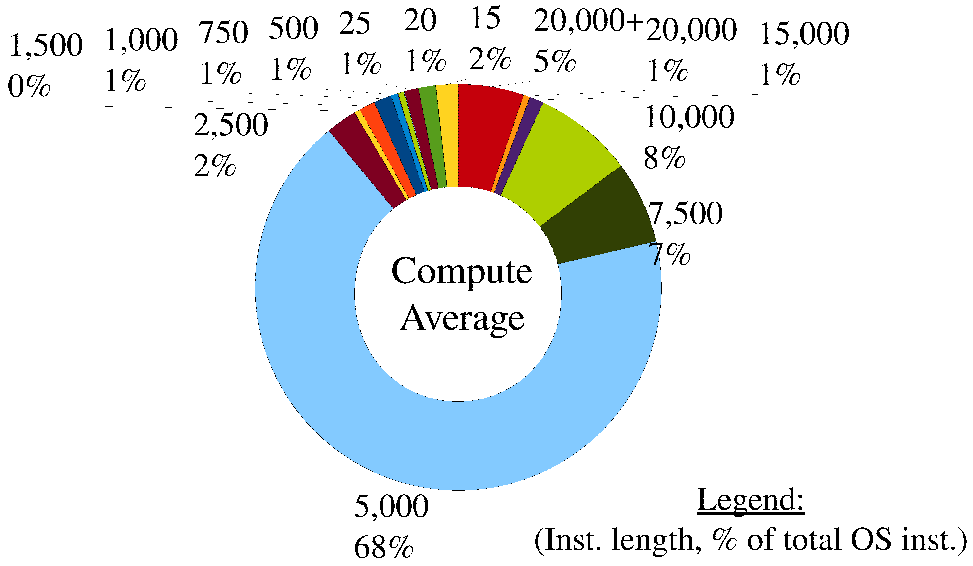

Figure 4.13. % of total OS instructions by invocation instruction-length for compute bound workloads.

gains via OS/User separation. Performance is clearly maximized with the lowest offloading overhead possible. If the core migration implementation is not efficient, it is possible that offloading may never be beneficial.

- **For any given offloading latency, choosing the appropriate switch trigger threshold N is critical.** When finding the optimal threshold N, two factors come into play, OS/User interference and cache-coherence. For very high N, we can not leverage the reduction of OS/User interference well. For very low N, cache coherence costs can offset any performance advantage seen from reduced interference. For example, Figure 4.9 shows that even with zero offloading overhead, moving from $N = 100$ to $N = 0$ substantially reduces performance. This is because the additional coherence overheads dominate the reduction in OS/User interference. We must therefore choose N dynamically at runtime based on a feedback policy, which in turn requires accurate predictions of OS invocation lengths.
- **Offloading short OS sequences is required.** We are somewhat surprised to see that maximum performance occurs when offloading OS invocations as short as 100 instructions long. This indicates that though long duration invocations dominate the total OS instruction count, short invocations have a larger impact on cache interference. 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 greatly complicates the OS developer's effort as nearly every system call is now a candidate for off-loading or offline profiling needs to be performed to exclude systemcalls that are infrequently called, regardless of execution length. Further, the overhead of the instrumented code is nontrivial for short routines. While the overhead of a few extra instructions is dwarfed by the overhead of migration, it must be noted that the instruction overhead is incurred every time the system call is invoked, regardless of whether it is off-loaded or not. This makes it clear that a software-based decision policy will incur high performance overheads and complexity if it strives to achieve the N with optimal performance. This reinforces our belief that a low overhead hardware prediction method for estimating OS run-length is

critical in optimizing offloading performance. It is worth pointing out that when you include sequences less than 100 instructions, the register rotation implementation for sparc is now a candidate for offloading. Because this is an extremely frequent event on the sparc V9 ISA, performance drops substantially due to the overhead of migration in all cases. Effectively using a remote cache as the backing store for the rotating register file does not make intuitive sense either. We did not explore additional work to exclude these rotations explicitly, however.

4.7.2 Offloading for Energy Efficiency

While our case study examines offloading for performance, alternate proposals examine OS offloading for energy efficiency. Energy efficiency improvements are directly tied to the amount of time that can be spent executing on lower power hardware. Table 4.3 shows the percentage of execution time that the OS processor utilized while running our server intensive benchmarks. An offloading threshold of $N = 10,000$ shows that even though run-lengths > 10,000 comprise 50% or more of the total OS instructions (as seen in Figure 4.12), they account for at most 17.68% of the total execution time. Lowering our offloading threshold to $N = 1,000$ almost doubles the execution time spent on the OS processor, even though sequences between 1,000 and 10,000 contribute a smaller portion of the total OS instructions. Clearly there is a nonintuitive relationship between OS invocations, percentage of instructions executed, and time of execution. While the majority of OS instructions are contributed by long running OS sequences, due to cache effects and coherence overheads, shorter sequences contribute a disproportionate amount to overall OS execution time. As a result, we believe it will be extremely hard for any intuition based offloading policy to achieve near-optimal performance.

4.8 Conclusions and Extensions

In this chapter we have seen that by offloading execution to a segregated core can help improve OS intensive workload performance by effectively segregating the cache occupancy of the OS and User cores. This solution comes with two major downsides, however. First, implementing a second OS core with private caches

Table 4.3. Percentage of total execution time spent on OS-Core using selective migration based on threshold N.

	Core Migration Threshold N				
Benchmark	**0**	**100**	**1,000**	**5,000**	**10,000+**
SPECjbb2005	38.84%	34.48%	33.15%	21.28%	14.79%
Apache	64.7%	45.75%	37.96%	17.83%	17.68%

requires substantial silicon area to provide a full second pipeline even if the relative cache area of each processor is equivalent to the original. Second, the overhead of performing a migration (mainly due to TLB shoot-down penalty) can easily erode the performance advantage gained through segregation. This means the proposed segregation technique will only be effective within a single socket CMP where the cost of a TLB shoot-down is low. In a many socket CMP where TLB shoot-downs must be broadcast to many processors on different sockets, the shootdown penalty will be too high to make this effective. Depending on the architecture one of the main costs of a TLB shootdown is not just the IPI for the TLB shootdown, but also the cache flush of dirty data that must occur on architectures without ASID compliant TLBs like x86. Next generation ISAs such as ARM however that support ASID have a significantly lower cost for TLB shootdowns (though they are still distributed via IPI), which should aid in improved migration costs. Because of this, we believe that having a second specialized core for OS execution may be valuable in single socket CMPs, which are becoming popular in tablets and mobile phones, but in future HPC systems a different solution is required.

In this chapter we identify that improvements in cache hit-rates are the primary driver for performance improvements and are offset by the high cost of migration. We know that these improvements are driven entirely by the memory system because we implement an identical performance pipeline on the symmetric OS core. Rather than try and devise solutions to reduce offloading overhead in HPC systems comprised of numerous cores and sockets, we believe a better solution is to simply eliminate the offloading overheads and try and achieve the benefits of offloading by simply segregating the caches on a single processor, effectively reducing the offloading penalty

to 0 in all cases. In the next chapter we propose several solutions for cache partitioning that improve performance without requiring additional silicon for an OS specific processor.

During the investigation of this dissertation an additional avenue of research for power efficiency has been developed by ARM for mobile processors that they call "BIG.little" as part of the ARM64 ISA. In this architecture they use two pipeline implementations, one out of order, one in order, that share a TLB and cache hierarchy and migrate between these two pipeline based on predicted up coming computation needs. This approach of utilizing a single cache simplifies the OS implementation (it simply does not know it is happening), but it also results in a performance loss for any execution thread (User or OS) that is executed on the little in order core. We view this work as complimentary to our caching work here because any improvement in power we might obtain when executing short sequences that do not take advantage of out of order pipelines could be symbiotic with the caching improvements to yet further improve the energy delay product of such a design.

CHAPTER 5

OS SEGREGATION IN LOCAL PRIVATE CACHES

Caches improve performance by reducing the number of times a processor accesses the relatively slow main memory. Effective caching decreases the off-chip bandwidth required by a running application, as well as reduces the latency in accessing memory. Out of order execution increases performance partially by hiding memory latency effects that are not handled well by caches. As in-order multicore chips return to the mainstream for throughput oriented applications, caches will play an even more important role in determining system throughput.

Modern operating systems use the concept of privileged execution to perform security and process isolation functions, and as a result many operations must occur within the OS. Every entry into the operating system must then be accompanied by a return to the user code and vice versa. Thus, by definition OS and User threads execute in a cyclic pattern, suboptimal for the traditional LRU cache policy. Table 5.1 shows the average number of instructions executed in both OS and User mode during workload execution. Because OS and User execution are strictly alternating, we can see that for Server workloads such as Apache and SPECjbb2005, the ratio of instructions executed within the OS is typically much higher than that of the HPC applications.

Cache thrashing occurs in nonshared caches when the working set size of the thread exceeds the capacity of the cache. Cache thrashing can occur in a shared cache, when two processors are simultaneously accessing it and the aggregate working size exceeds the capacity of the cache. Similarly, thrashing can occur when two independent processes/threads are alternately accessing a nonshared cache. High performance

Table 5.1. Average length of execution before switching modes (in instructions).

Benchmark	OS Execution	User Execution
Apache	329	156
SPECjbb2005	692	1,770
hmmer	89	33,280
mummer	20	47
fasta_protein	714	14,862
blackscholes	2,098	158,828
canneal	18	120
mcf	15	22

computing typically utilizes a one-to-one mapping between threads and processor cores because competing threads and context switching of N-to-one mappings usually results in suboptimal throughput.

Table 5.2 shows us that even what is commonly considered a single thread of execution, operating system invocation can be the dominant cause of evictions at all levels of the cache for both Server and HPC workloads. We track this by precisely measuring the state of the processor when a cache eviction occurs. For example, if a L1D eviction was caused, we check if the processor is currently executing a priviledged instruction or user instruction that has generated the data fetch causing the eviction. Similarly for the L1I, if the instruction that misses the instruction cache is a priviledged instruction, then we deem the subsequent eviction as cause by priviledged mode execution. Server applications have high levels of OS utilization and therefore have higher OS-related cache utilization than HPC applications. Non-OS intensive workloads, however, are also affected by the alternating nature of OS execution. For many of these applications, which alone exhibit good cache behavior, OS invocation can be the dominant cause of cache misses. While no cache can eliminate compulsory misses, cyclic OS invocation allows us to investigate techniques that could possibly mitigate misses due to OS invocation, an infrequent but dominant source of conflict misses even in HPC applications.

Operating system code has significantly different characteristics than user code. Table 5.1 shows that OS execution, while frequent, is often much shorter in duration than user execution. This variability in run-length (both in the OS and user execu-

Table 5.2. Percentage of cache misses caused by OS execution.

Benchmark	L1I	L1D	L2
Apache	72.67%	73.43%	55.10%
SPECjbb2005	71.65%	74.92%	67.57%
hmmer	60.41%	4.25%	4.10%
mummer	78.88%	42.05%	10.72%
fasta_protein	48.20%	31.21%	90.07%
blackscholes	85.53%	8.99%	24.71%
canneal	83.61%	43.73%	11.48%
mcf	79.59%	19.24%	1.37%

tion) is application-dependant, and there does not appear to be a typical pattern of OS usage even among compute bound applications.

The average run length of both user and operating system code indicates that among both HPC or server applications, there is significant variability in this cyclic relationship. Figure 4.2 showed the percentage of invocations that are spent in instruction sequences of varying lengths for both the OS and user application. The majority of operating system invocations are exceptionally short where as 10,000+ instruction sequences tend to dominate only the HPC application execution. Due to the very low amount of operating system execution in the HPC applications however the data are slightly misleading; the total number of long OS invocations is actually lower than in either Apache or SPECjbb2005, but as a fraction it looks very high.

The variability in OS execution was broken down further in the previous chapter in Table 5.2 showing the percentage of misses caused in each cache level by operating system execution. L1 caches tend to be affected more by short duration accesses of the OS than the L2, which is more capacity driven. Long OS executions, such as reading and writing of data on behalf of the user application, dominate the L2 evictions. If these data were reused by the user code, these OS evictions would not be costly in terms of application throughput. Table 5.3 shows that for most applications, the OS and User application have large distinct memory footprints, even those that are compute bound.

By identifying that the OS and user application are effectively competing for cache resources much like two independent threads, we investigate three techniques

Table 5.3. Memory footprint of execution (in MB).

Benchmark	Total	User Only	OS Only	User/OS Shared
Apache	15.86	6.34	9.02	0.50
SPECjbb2005	150.46	108.98	20.13	21.34
hmmer	12.53	8.16	1.08	3.28
mummer	120.83	73.84	1.04	45.94
fasta_protein	29.83	1.51	23.03	5.28
blackscholes	8.93	1.84	6.97	0.11
canneal	37.78	29.53	8.24	0.00
mcf	6.07	5.903	0.162	0.00

for exploiting OS/User virtual threads in the context of single threaded execution. Previous investigations have attempted to identify program phases and adapt cache resources appropriately [82]. To our knowledge this is the first work to treat single threaded applications as two virtual threads and view them as competing for what has been thought of as a nonshared resource. In this paper we look at *cache segregation*, *way-partitioning*, and *insertion policy* in private caches to determine if OS/User interference can be alleviated. We present our methodology before continuing on to the proposed solutions.

5.1 Experimental Methodology

5.1.1 Caches

We use an execution driven simulator so that the OS execution can be simulated with full accuracy. For all our simulations Table 5.4 shows the baseline cache parameters. Modifications to this cache are described in the relevant sections. Our timing parameters were obtained from CACTI 6.0 [79] targeting a frequency of 3.5GHz. At this frequency an optimistic memory latency of 350 cycles is used in all experiments (based on real machine timings from Brown and Tullsen [39]). A slower main memory assumption of 500 cycles would further augment the performance of the proposed solutions by approximately 2%.

5.1.2 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

Table 5.4. Baseline cache configuration.

L1 Inst.	16KB, 64B line, 2-way LRU, 1 cycle
L1 Data	16KB, 64B line, 2-way LRU, 1 cycle
L2	1MB, 64B line, 2-way Banked, 16-way LRU, 12 cycle
Memory	350 cycle uniform latency

the PARSEC, BioBench, and SPEC-2006 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, and SPECjbb2005 comprise our server oriented workloads. All benchmarks utilize a 1:1 mapping between cores/threads with four independant cores being simulated except Apache which self tunes thread counts to optimize throughput. 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 have summarized these results and note any outlier behavior.

5.1.3 Processor Model

We use Simics 3.0 to drive our execution based simulations based on an in-order ultraSPARC core. 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 multithreading (for example, the Sun Niagara and Rock designs and the recent Intel Atom design) [16]. While out-of-order (OOO) cores hide memory system latency better, the cache optimizations presented here should be viewed as a design alternative to OOO execution, which is extremely energy inefficient.

On the SPARC platform, determining if you are executing within the OS is easily achieved by looking at the PSTATE register. All our proposed techniques use this register to determine what is running as part of the operating system virtual thread and what is running within the User virtual thread. Via this definition, system calls that occur in privileged mode but within the user address space are captured as part

of our OS behavior as well as functionality that occurs within the kernel address space.

The SPARC ISA has several unique features that cause many of the short duration (<25 instructions) OS invocations. Because these short duration OS invocations have a relatively small impact on cache thrashing, we believe our results would be similar on alternative ISAs such as x86, alpha, or ARM.

5.2 L2 Bank Partitioning

The simplest way to reduce the competition for a shared resource is to remove one of the competitors. For OS and user execution this means routing their individual requests to a subset of an existing cache, thereby eliminating 100% of all interference. Static partitioning of cache resources in a 2-way banked cache effectively halves the cache capacity seen by both the OS and user, however, so any capacity based segregation must overcome the effective loss of capacity.

The basic idea is simple. Just as the use of separate instruction and data caches is common-place today, we believe that a separate OS cache per core may be worthwhile. For the experiments in this section, our uniprocessor design implements traditional split L1 instruction and data caches that are shared by both OS and User code. We architect a solution that uses the existing banked L2 design to isolate L2 references from OS and user execution.

5.2.1 Block Lookup

When partitioning the L2 cache, 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 examined a solution that allows blocks to be replicated in both banks, but the benefits were nearly nonexistent. Additionally, this required cache coherence between banks negates the simplicity of our proposal. While our cache is mutually exclusive at the block level, the OS and user applications do share a nontrivial amount of data (though not typically instructions). Table 5.3 shows the total application footprint in main memory broken down by accessors. Therefore, we must implement a lookup scheme that can determine if either the OS or User bank contains the requested block.

When an L1 miss is encountered, one of the two L2 cache banks may contain

the data. In parallel lookup mode, both cache banks are simultaneously accessed. On an L2 miss, the newly fetched block is placed in the OS-bank or user-bank of the L2 based on our block placement policy (discussed subsequently). Alternatively, on an L1 miss, we can predict which bank is likely to contain the data and look it up first. If data are 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, and 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 power than the parallel lookup mode. The L2 access time for serial lookup is equal to the baseline on a correct prediction but increased on mispredictions. The motivation for serial lookup is that our banked design will consume less power than the parallel lookup mode if we can perform accurate bank prediction.

5.2.2 Block Placement

We examined three different block placement policies. The first places all data blocks that are fetched by privileged instructions into the OS-bank and everything else into the user-bank (designated as "Data Only" in Figure 5.1). 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 "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 "Instructions and Data").

5.2.3 Bank Prediction

For bank prediction in the serial lookup policy we employ a predictor that mirrors the block placement policy. For example, if the instruction is privileged and our block placement policy is "Data Only." We first look for these 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 instruction block lookups. It only incurs mispredictions when the OS or user attempts to access data that were first brought

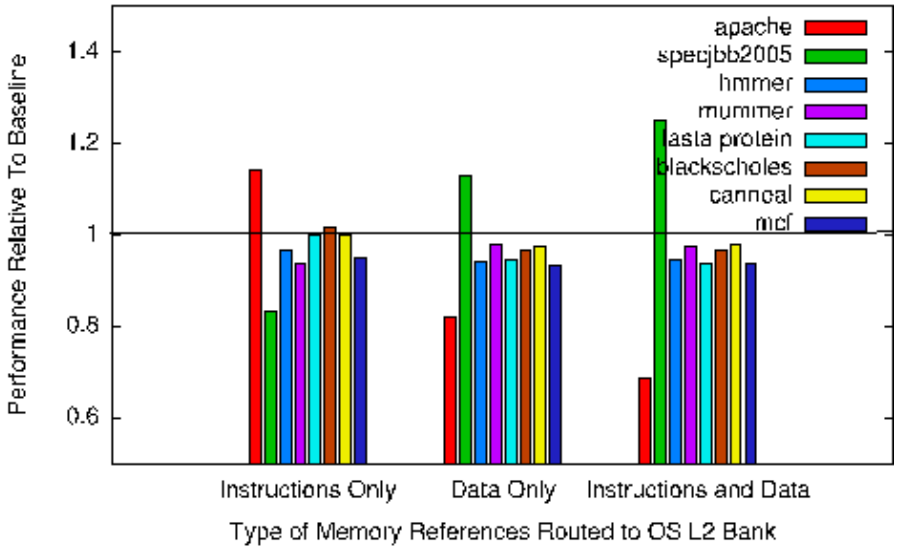

Figure 5.1. Performance of OS/User L2 bank segregation with parallel lookup.

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 [83]. Our simple predictor provides an average accuracy of 99.9% for Instruction Only routing, 81.2% accuracy for Data Only routing, and 87.3% for Instruction and Data routing policy. We did not consider more sophisticated bank predictors in this study due to the high accuracy of this scheme that requires zero storage overhead.

5.2.4 Evaluation

For this set of experiments we assume that a parallel lookup takes 9 cycles. Serial lookup takes 9 or 12 cycles according to CACTI (since we must route the request to the predicted bank, perform a tag lookup in the predicted bank, discover a miss and route the request to the second bank, and then finally perform tag and data lookup over the 512KB bank). Thus parallel lookup is no slower than our baseline cache but requires more energy per access. Serial lookup is equally efficient in terms of power when there is a correct bank prediction but slower and requires more power when there is an incorrect prediction due to the second set of tag lookups.

We are able to achieve a performance improvement of 25.1% for SPECjbb2005 and 14.4% for Apache as seen in Figure 5.1. 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 sometimes seen by separating only the OS data or instructions references, allowing the other to share space with user blocks. It is easy to determine the optimal routing policy for each workload based on performance feed back, using the L2 miss rate and a static sampling window of 10 million instructions for each policy.

Thanks to the high bank prediction accuracy, we observe that serial lookup has performance very close to that of parallel lookup. Instruction Only routing achieves equal performance to parallel lookup because prediction accuracy is 99.9%. Data Only and Instruction and Data routing can achieve best performance of 12.7% for Apache and 22.1% for SPECjbb2005 utilizing serial lookup.

- *Compute Bound Performance* For the compute-intensive applications, we observed that all 3 of our examined bank placement policies yield anywhere from 7.2% slowdown to 1.7% performance improvement compared to the baseline. On average compute benchmarks see performance degradation of 3.8%. This is not surprising; user and OS execution are limited to half of their baseline capacity. This is poor allocation for compute-bound applications (and many others) that engage in little OS activity. This underutilization of the OS cache is extremely detrimental to compute bound workloads that are performance bound by cache capacity. To remedy this common case, we require only a few hardware counters that allow us to determine if fewer than 10% of all instructions in the last 1 million instruction epoch were privileged instructions (in other words, there was little OS activity). If so, the bank placement policy can use either bank to place a newly fetched block like the baseline resulting in identical performance for the compute bound applications.
- *Cache Size Sensitivity* The issue of static capacity allocation via a banked design is significant. While the workloads used in this study appear only mildly sensitive to cache capacity when moving from 1MB to 512KB, there are certainly workloads which exhibit high sensitivity. HPC in particular tend to be capacity sensitive, and for those workloads we expect the alternate bank

placement policy to be required to achieve performance equal to the baseline. Even server workloads which have a high OS utilization must overcome the capacity reduction in a banked placement policy. Table 5.5 shows a 2-way banked design that implements OS/User segregation at a variety of L2 sizes (L1 sizes and policies the same across all experiments). Below 256KB performance is lost when performing static cache segregation.

While this proposal leverages the natural banked design of the L2 cache, it is possible to implement banking in caches of any size. We explored several organizations that adopt an L1 (I&D) OS cache, but found negligible performance improvements. The small size of L1 caches can not be subpartitioned without incurring a substantial performance penalty due to increased capacity misses as seen in Table 5.5.

5.3 Variable Insertion Policy

Both bank and way-partitioning are able to achieve performance improvements for our server based workloads by partitioning cache resources to isolate interference between the OS and User virtual threads. A recent proposal by Jaleel et al. [49] has suggested that in shared caches it is possible to dynamically allocate capacity between two competing resources without statically limiting capacity. By modifying the insertion order of a newly fetched line so that it is not always inserted into MRU position, one can shorten the expected cache life of a reference that is not expected to be reused.

Adaptive insert allows lines that are inserted in non-MRU position to be promoted to MRU position should they be referenced again before being evicted from the cache. Adaptive insert has an advantage over way based partitioning because it does not inherently limit the capacity or associativity of each partition. Thus, should a set exhibit high cache locality, even references inserted into LRU position are able to migrate to MRU position, increasing the set's effective reuse reach.

This approach is widely regarded as the state-of-the-art low-complexity approach for shared cache partitioning – hence, we examine its applicability here. Given the success of our partitioning proposals in Sections 5.2 and 5.4, one might think that a variable insertion policy should also see significant performance improvements for our

Table 5.5. Maximal performance relative to baseline policy at varying L2 cache sizes.

Benchmark	128KB	256KB	512KB	1MB	2MB
SPECjbb2005	0.825	1.023	1.1217	1.251	1.245
Apache	0.927	0.991	1.092	1.144	1.158

server workloads. With our implementation we could not achieve such performance. Before discussing our conclusions on why we could not, we describe our methodology.

5.3.1 Implementation

We implemented a fixed insertion policy across all sets, much like we did in Section 5.4, allowing that global policy to be adjusted dynamically based on performance feedback. During a cache insert the processor is polled to determine if OS or User execution is occurring. Based on the virtual thread executing, we consult a 4-bit partitioning register that determines where in the LRU ordering this new cache block should be placed. By maintaining a constant insertion policy across all sets, this insertion policy requires only 8 bits of total storage overhead. A global policy also requires no additional storage overhead for determining the optimal insertion partitioning. Figure 5.2 shows how cache block replacement and insertion work for a given insertion policy. Adaptive insert requires no changes to the cache line update policy or the victim selection policy.

It is possible to allow each set in the cache to maintain its own local insertion partitioning. This would require 8-bits per set or 16kB of storage overhead for our 16-way 1MB L2. While not unreasonable, maintaining performance information about these sets would require at least several more bits per line as well. These performance bits are required to perform dynamic optimization of the insertion policy based on a feedback mechanism similar to that described in Section 5.4.4. Maintaining this state information also results in exponential growth of the search space making optimal partitioning infeasible. We did not explore such an implementation because the design complexity in real hardware would be too great.

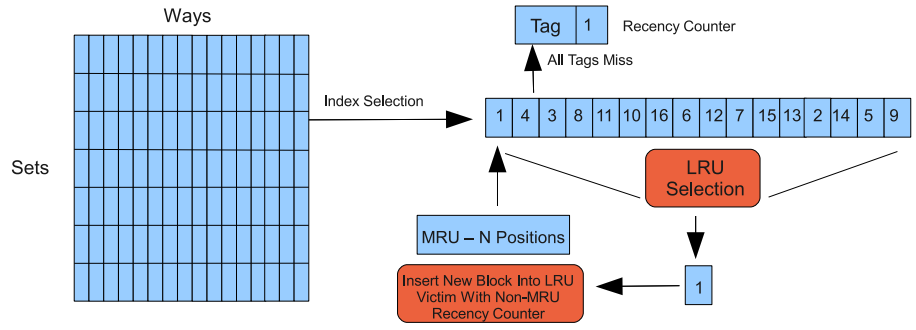

Figure 5.2. Variable LRU stack insertion.

5.3.2 Block Lookup and Placement

Adaptive insert varies from previous proposals in how we lookup blocks and choose our insertion policy. Block lookup is identical to traditional cache lookup. For both User and OS virtual threads, blocks will be inserted into the LRU victim's location, but with a modified LRU ordering.

Adaptive insert still allows us three choices for block insertion. We can use the OS insert placement for both Instructions and Data, just Data, or just Instructions. For this experiment we choose to examine only Instructions and Data insert placement, meaning that all OS and User references will be adaptively inserted according to the partitioning.

5.3.3 Performance

We had hypothesized that we would see similar performance improvements compared to our best-performing partitioning schemes. However, we find Apache is able to achieve less than 1% performance improvement, while SPECjbb2005 achieves a modest 6.8%. For the majority of insertion policies the performance of both server workloads remains within 1% of the the baseline LRU policy. To understand why we were not seeing performance improvements in line with way-partitioning we had to revisit the underlying mechanism on which adaptive insert is based.

Adaptive insert is able to reduce the number of misses that occur in a cache by not allowing cache lines with a low likelihood of reuse to replace those with a high likelihood of reuse. One specific example of this is capacity thrashing. Consider a cache set with associativity N that is handling N+1 blocks that are being accessed

sequentially in a loop. This cache set incurs a miss on every access. The cache set has a reuse reach that is less than the reuse distance of the application. Adaptive insert can be used in such a situation to insert lines into MRU (or any non-LRU) position, which effectively allows some lines in the set to remain in the cache for potential reuse.

User/OS virtual threads execute in a strictly alternating pattern. They also tend to have disjoint working sets with very little shared data between them and no shared instructions. Set partitioning is effective because it allows some fraction of each thread's working set to remain in the cache indefinitely, never being evicted by the other. Adaptive insert extends the reuse distance of a cache set, but does not guarantee that some resources of each virtual thread will remain in cache. Any line inserted in non-MRU position can still be promoted to MRU if a cache hit to this line occurs. Thus, even though the OS/User virtual threads are exhibiting cyclical behavior the execution of each has enough temporal locality that adaptive insert is not able to overcome the thrashing behavior.

5.4 L2 Way-Partitioning

In Section 5.2 we saw initial evidence that partitioning operating system and user execution in the cache hierarchy can result in significant performance improvements for OS intensive workloads. Bank partitioning, while easily implemented, has several drawbacks. Foremost, banks usually constitute large portions of a cache, and it is unlikely that the optimal partitioning of cache space between OS and User virtual threads will fall exactly on the bank partition size that is fixed at design time. This limitation can be overcome by allocating space within the cache via a different mechanism, way-partitioning.

In highly associative caches, way-partitioning allows OS-User segregation by dynamically allocating N of M ways to the OS virtual thread and $(N - M)$ of M ways to the user virtual thread. By allocating a subset of the total ways to each virtual thread, we can change the granularity of resource partitioning as finely as the associativity of the cache allows. For this work, we examine a 16-way L2 cache that allows 15 possible subpartitionings.

Way-partitioning differs from bank partitioning not only in the partition granularity but in the associativity available to each partition. Bank partitioning in Section 5.2 statically allocates 50% of the capacity to both virtual threads, but the associativity of those banks remains 16, the full associativity of the cache. Way-partitioning always reduces the associativity of each subset, a negative aspect of this technique. For example an 8/8 way allocation between virtual threads is likely to have worse performance than a 50%-50% bank partitioning that maintains 16-way associativity.

5.4.1 Implementation

For this study we implement way-partitioning through victim selection within a standard LRU replacement policy similarly to column caching [84]. Column caching works by choosing a LRU victim only from those ways specified in a bit mask of victim information. We use way allocation by either the OS or User as our criteria bitmask for our victim selection. Tracking of this allocation is done by a single bit per L2 way that is set to 1 if this is deemed an OS way and 0 if deemed a User way. The storage overhead of OS/User way-partitioning is only 16 bits when allocating ways across all sets, opposed to allowing variable way allocation within sets where the storage requirement would be 2kB. We describe the three phases of cache behavior.

5.4.2 Block Lookup

Just as in Section 5.2 we chose to make the decision that mutual exclusion will exist between the contents of the User and OS partitions. While shared blocks could be replicated between the partitions, this would result in significant changes to a cache design to maintain coherence. Because the User and OS threads can share data, we must implement a lookup scheme that can determine if either the OS or User partition contains the requested block.

Because a data block can exist in either the User or OS ways, block lookup can utilize existing cache hardware with no change. Just as in the baseline policy, all tags are examined for a match in parallel. This guarantees a hit (if it exists) in the same amount of time as a traditional LRU scheme utilizing no additional power. This is the lookup scheme used for the results in this dissertation chapter.

While parallel tag lookup guarantees a tag match using equal power consumption

to the baseline LRU cache, there is potential for power savings when implementing way-partitioning. Because the majority of accesses are to nonshared blocks, we are able to achieve high prediction rates on where a block is likely to be stored. While not the focus of this paper, it would be possible to overload the victim selection bitmask for use as a way prediction technique to decrease the cache access energy with the techniques described by Powell et al. [85].

5.4.3 Block Placement

While block lookup occurs across all ways, way-partitioning must only allow OS/User insertion into a defined number of ways within the set. We implement partitioning by slightly modifying the LRU victim selection algorithm as shown in Figure 5.3.

The only change required to a traditional LRU design to implement OS-User partitioning occurs during the selection of a cache line replacement. When no tag is found to match in a set, we have a cache miss. LRU policy typically selects the LRU line from the set as the destination for the replacement to be inserted (and set to MRU access time). Victim selection occurs by filtering the lines in the set by their OS-User tag bit before performing LRU ordering. For example, if the operating system is executing and a cache miss occurs, LRU selection only occurs on those lines that have the OS bit set. As a result, LRU ordering is maintained within the OS-User partition without changing global LRU update functionality.

For a LRU policy when a hit occurs to a cache line, the accessed time for that line is logically updated. In our implementation, LRU ordering occurs on a global level among all ways just like traditional LRU. Maintaining a global LRU is equivalent to maintaining local LRU per OS/User thread no matter what order the partitions occur in or what fraction of the ways are allocated to each partition.

5.4.4 Dynamic Allocation of Ways

There are many existing proposals for the most efficient method of allocating ways between competing threads. Two prominent ones include (i) allocating certain ways across all sets to one thread and the rest to other [84] or (ii) allocating N ways of each set to one thread, where N varies per set [46]. The latter provides the highest

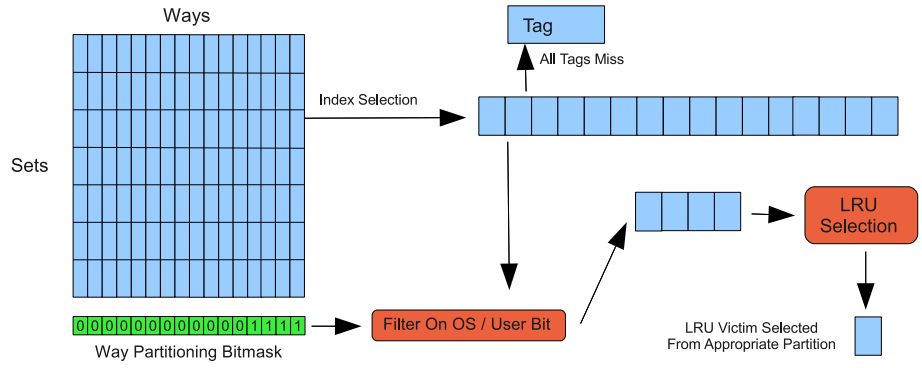

Figure 5.3. Victim selection in OS/User way-partitioning.

resolution partitioning possible but comes at the expense of per set counters or other nontrivial monitoring hardware. We chose to implement way allocation across sets (proposal (i)) to limit the complexity of our proposal.

We use a feedback based approach to select the optimal partitioning based on algorithms described in prior work [81]. To determine the optimal allocation, we monitor L2 Misses/1K Instructions (MPKI) using a traditional LRU policy, no way-partitioning, and a sampling epoch of 5 million cycles. We then choose an initial partitioning that is closest to the percentage of total instructions that are being executed by the OS. This initial partitioning, plus two partionings both above and below the current partitioning ratio, are executed serially, each for the sampling epoch. If none of these partitions reduce MPKI compared to the baseline LRU policy, we immediately fall back on LRU policy resulting in no change in IPC compared to baseline for the remainder of execution.

If one or more of these five samples reduces the L2 MPKI, the partitioning with the lowest MPKI is chosen and run for an execution epoch of 25 million cycles. We then sample one partition on either side of the current way-allocation, serially, for the sampling epoch. If neither sample reduces MPKI from the present MPKI we maintain our current partitioning and double the execution epoch. If one or both sampled partitionings reduces the MPKI, we modify our current partitioning and reset the execution epoch to 25 million cycles. Using this technique, we find that convergence on optimal partitioning often occurs within 50 million cycles and that a sampling epoch of 5 million cycles appears to be quite resilient to changes in program

phase, resulting in near constant way-allocation throughout program execution.

5.4.5 Performance

To evaluate the effectiveness of OS/User way-partitioning we evaluate all possible subpartitions for our workloads in Figure 5.4. The results show a significant difference in performance for Apache and SPECjbb2005 for a given partition allocation. Apache achieves a maximum performance improvement of 4.5% at way allocation 11-5 (OS-User), while SPECjbb2005 is able to achieve a 41.6% improvement by allocating only 3 sets to the OS virtual thread. These optimal allocations align very closely, however, with the relative percentage of OS/User instructions executed by each thread. Straying too far from this way allocation results in performance degradation in both workloads.

Figure 5.5 shows the average performance for compute-bound applications as well as the standard deviation from that average. Reserving a small number of ways for the operating system can achieve an average performance within 1% of LRU for compute bound applications with less than 1% standard deviation. For some compute bound applications modest speedups over LRU are achievable; however, that performance increase is not representative. OS/User way-partitioning is never able to achieve an increase in average performance for the compute bound applications.

5.4.6 Instruction and Data Routing

Our basic way-partitioning algorithm is able to improve performance over the baseline LRU policy through fine grained allocation of cache resources. Bank partitioning has provided some clues that segregation of instructions and data both may not be optimal even within fine grained way-partitioning. Just as we examined three different block placement policies in the banked cache design in Section 5.2.2, we now model two additional policies on top of way-partitioning. The first routes only *instruction* blocks fetched by the operating system into the OS partition. The second places only *data* fetched by the operating system into the OS partition. The previously discussed "Instruction and Data" routing shown in Figure 5.4 is included again in each graph for comparison. Figure 5.6 shows the performance of our 3 block placement policies across the various way-partitioning allocations. The compute average performance

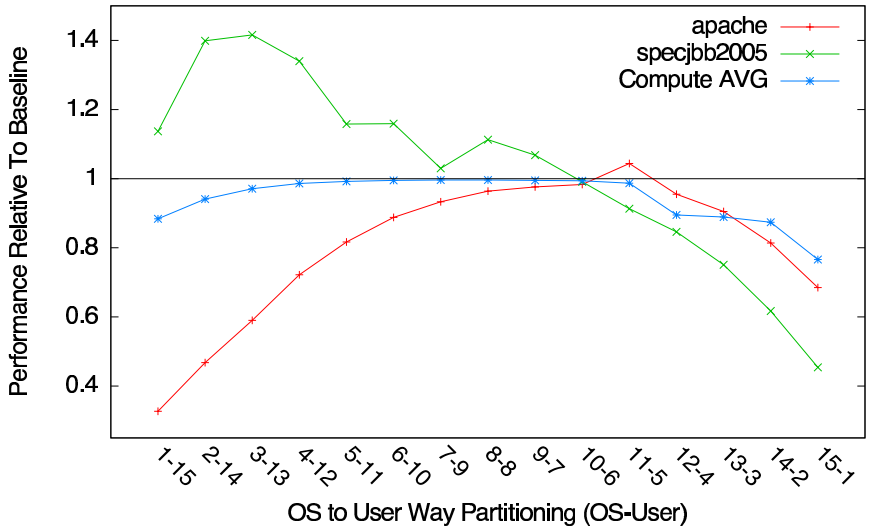

Figure 5.4. Performance of OS/User way-partitioning at various thresholds.

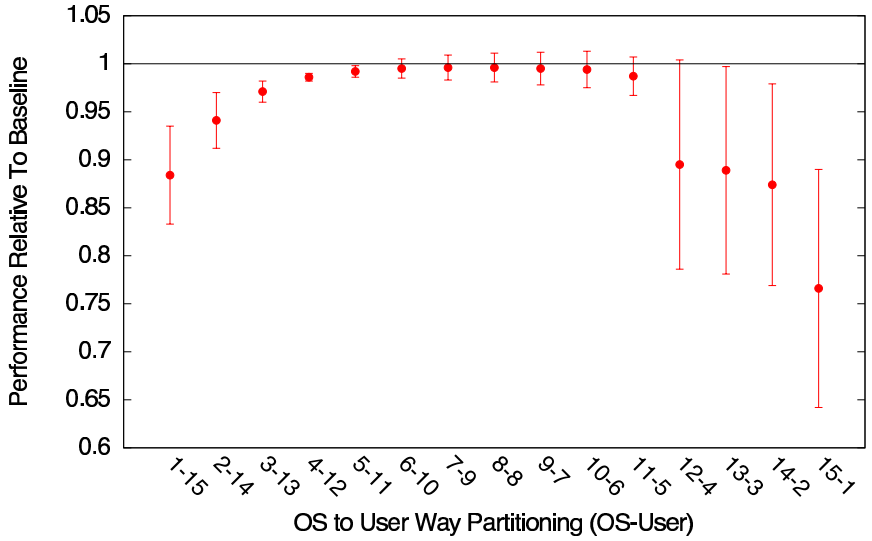

Figure 5.5. Average performance and standard deviation of compute bound workloads using OS/User way-partitioning at various thresholds.

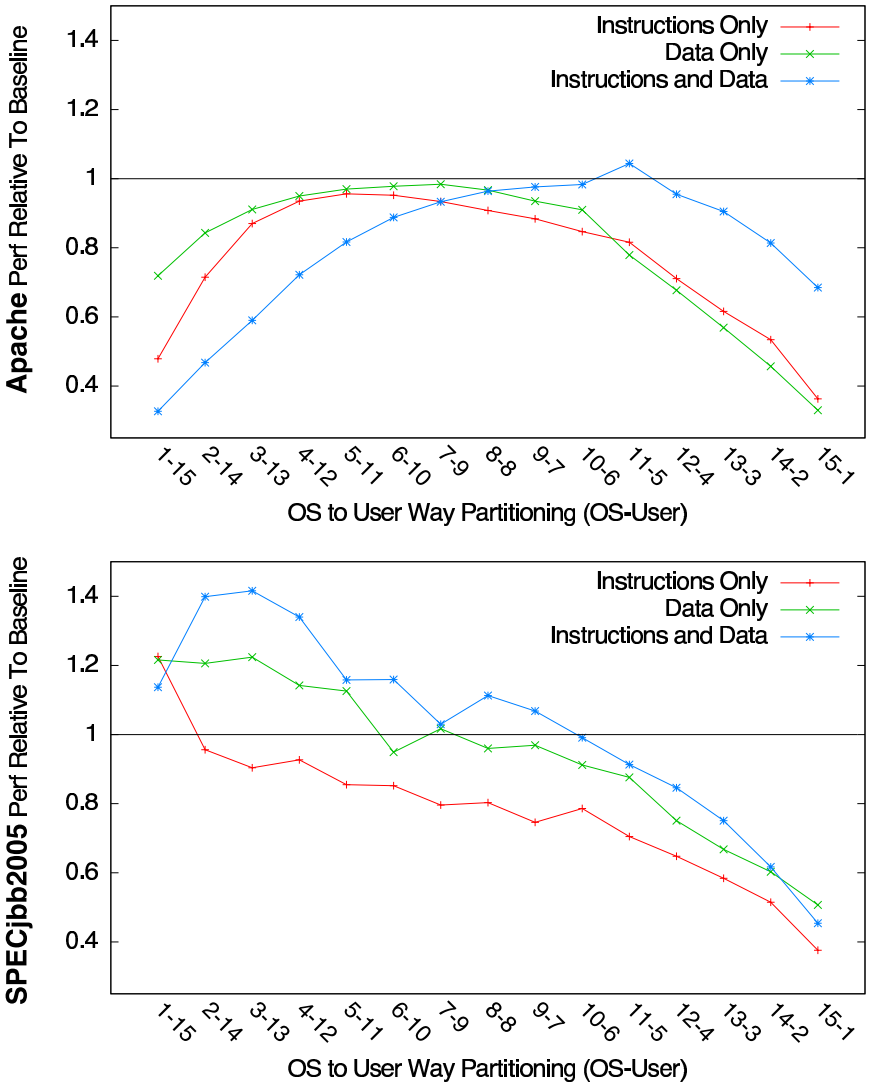

Figure 5.6. Performance of block placement policies with OS/User way-partitioning.

is not shown; both Instruction-only and Data-only track the performance trend of Instruction and Data routing very closely, never achieving improved performance.

We had hypothesized that Apache would show its best performance using Instruction only routing, just as it had in cache segregation. Instruction only and data only block placement policies were never able to achieve a performance improvement compared to baseline. Both of their performance peaks are also skewed significantly towards a User partitioning dominance compared to Instruction and Data Routing. It appears that within Apache the OS and User virtual threads are both sensitive to capacity, as are the block routing policies.

SPECjbb2005 achieves maximum performance for Instruction and Data routing but also achieves good performance for Data only Routing. Instruction-only routing typically underutilizes the OS partition designated. Only at the smallest OS partitioning level does Instruction-only block placement achieve the best performance, indicating that there is some kernel of OS instruction references that tend to be evicted in a LRU policy, but are cyclically executed by the OS.

5.5 Comparing Segregation Designs

High performance computing has traditionally been dominated by single threaded applications that are very compute intensive. Only in the last 15 years have web-serving, databases, and other throughput oriented tasks become a significant portion of what is considered high performance computing. This paper shows that for these server based workloads, the operating system execution within these workloads causes significant cache interference with user execution. We propose that for these server workloads, it makes sense to logically decompose single threaded applications into two virtual threads of execution. By treating these virtual threads as competing resources in nonshared caches, we can dynamically allocate cache resources between our two virtual threads. For server workloads, this cache partitioning results in an aggregate speedup for what is traditionally thought of as a single threaded application. In this work we investigate three low cost techniques for decreasing interference between the proposed OS-User virtual threads.

5.5.1 Bank Partitioning

As caches grow in size, circuit designers are motivated to implement banking to decrease cycle time and reduce cache access power. A typical banked access policy is to alternately route odd and even addresses between banks (referred to as word interleaving). We propose that banks can alternatively be partitioned between OS-User virtual threads. Bank partitioning in a two-bank cache reduces the cache capacity available to each virtual thread by one half, but maintains the full cache associativity. We also dynamically resort to a word-interleaved banked organization for workloads with little OS activity. With such an approach, we are able to achieve speedups between 14.4% and 25.1% for server workloads without any performance

degradations for HPC workloads that do not segregate well into OS-User virtual threads.

The major deficiency of bank partitioning is that server workloads, with high OS utilization, rarely exhibit their highest performance when allocating cache space equally between OS and User virtual threads. To maintain the power efficiency characteristics of a banked cache design, bank prediction and serial lookup of banks must be implemented, increasing complexity.

5.5.2 Way-Partitioning

Modern caches are growing in associativity with 16-way caches becoming commonplace. These highly associative caches can be partitioned by allocating a subset of ways to each OS-User virtual thread. This allows for a much finer grained control of the cache space allocated to each thread. However, the fine grained capacity control comes at the expense of decreased associativity. The effects of reduced associativity can be seen by examining a 50/50 way-partitioning in which each partition has an associativity of 8, compared to 16 in the 50/50 bank partitioning. The latter outperforms the former at this capacity allocation.

Way-partitioning can occur within sets, where each set is allowed to dynamically adjust its partitioning, or across sets, where all sets are partitioned equally. The former is likely to yield higher performance but has a significantly higher implementation complexity than the latter. We implement way-partitioning across sets and use online dynamic feedback to choose an optimal partitioning. Our implementation is able to achieve best performance of 4.5% and 41.6% for Apache and SPECjbb2005 while again mitigating any performance loss for HPC applications. Way-partitioning allows finer granularity for capacity allocation, but sees performance reduction due to decreased associativity.

5.5.3 Adaptive Insert

Adaptive insert is an alternative to way-partitioning for shared caches. It works by inserting newly fetched lines into a set in non-MRU position. By tuning the LRU ordering position at which new lines are inserted, one can effectively limit the cache space of threads that are not LRU friendly and expand the reuse reach of those which

are. Adaptive Insert does not statically limit the capacity of either competing thread. Should an application temporarily exhibit good locality, cache blocks it has allocated are allowed to enter MRU position, thus increasing its allocated capacity.

Our implementation of adaptive insert is not able to achieve the same level of performance as seen by either bank partitioning or way-partitioning, as shown in Figure 5.7. We believe this is due to the cyclical execution that OS-User virtual threads exhibit. In a shared cache, where adaptive insert has been shown to be effective, both threads are concurrently executing and inserting blocks into the cache at some nonzero rate. In OS-User virtual threading, only one thread is executing at a time. Because of this, it is possible for one thread to evict all blocks of the alternate thread before execution returns. This is effectively the same behavior with the baseline LRU policy. As a result, performance of adaptive insert has little variation from the baseline LRU policy for both server and HPC applications.

5.6 Conclusions and Extensions

Separating OS intensive workloads into OS-User virtual threads and segregating their execution within a private cache yields significant performance improvements. There are many ways that this segregation can be leveraged. We have investigated three techniques, bank partitioning, way-partitioning, and adaptive insert, in this paper. Bank partitioning and way-partitioning of OS-User virtual threads can speed up server workloads by as much as 14.4 and 45.1% depending on the implementation. We identify an inherent property of OS-User virtual threading that we believe makes adaptive insert less effective than static segregation.

To limit hardware complexity, we implement partitioning across all sets, rather than within each set, for both way-partitioning and adaptive insert. While more complex, it is possible that we will see additional improvements by allowing resource allocation to occur individually at each set. The associativity reduction of way-partitioning has a detrimental effect on performance. Exploring alternative partitioning designs that simultaneously allow fine grained capacity allocation without reducing associativity are extensions we intend to explore as part of our future work.

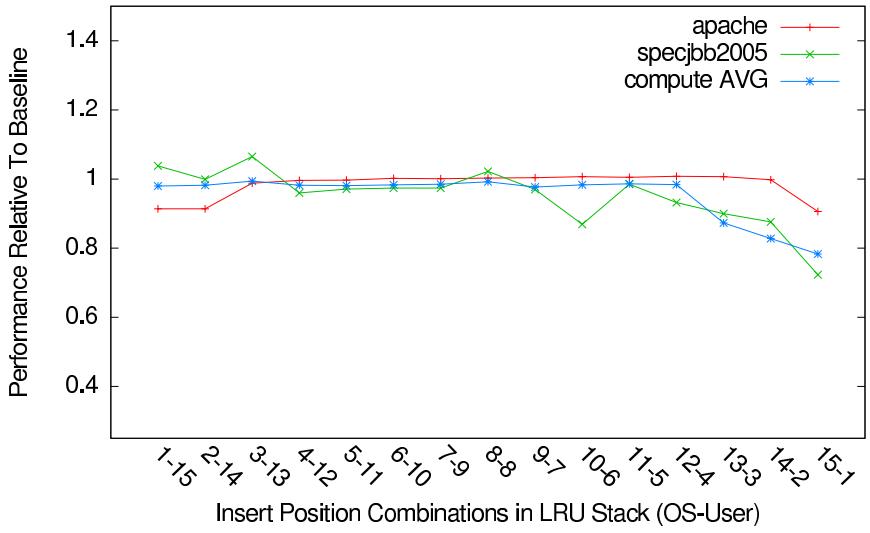

Figure 5.7. Performance of OS/User insert policies at various thresholds.

CHAPTER 6

OS CO-DESIGN FOR MULTILEVEL CACHE HIERARCHIES

In the previous sections, we have looked at the possibility of offloading OS execution in hardware to a specialized core containing private cache resource without any OS intervention. Due to the added expense of an additional core where the user core goes unutilized, this approach, while improving performance, does not make sense from an energy efficiency standpoint. We have also looked at the possibility of segregating a single core's private caches such that the OS and user runtime execution do not interfere with each other when executing in cyclical fashion. This reduces both the overhead of control flow transfer to an additional core and does not require overprovisioning of chip area that will only be used in mutually exclusive execution phases. Neither of these proposed solutions is ideal. Offloading execution to an alternate core is a heavyweight operation to perform without OS intervention and knowledge. Private cache partitioning can be beneficial, but the window of partition sizes for which application performance improves rather than degrades is small, and in practice it may be hard for an online feedback mechanism to appropriately match partitioning sizes to variable execution. When these partition granularities do not match, performance is left on the table.

In both these experiments, we focused on improving the performance of a single core that is executing one of the many threads of execution that are occurring on a modern chip multiprocessor or multisocket machine with only moderate success. In this section we examine another aspect of system performance that is heavily influenced by operating system and hardware execution, scaling performance up across multiple cores and sockets in multiprocessor machines. While speeding up

single core execution is an important problem, single thread performance has largely stalled due to the lack of instruction parallelism that can be further extracted from a thread, limited by power efficiency. Simultaneously, cache hit rates are high enough that large improvements are becoming harder and harder to achieve. As a result, processor designers have turned to chip-multiprocessors (CMP), which require many executing threads to scale up aggregate performance. These CMPs introduce a new problem of not just operating system execution but coherent operation of serialized operating system execution among multiple concurrently running processes.

In the last 10 years this has come to the forefront of HPC with 2, 4, 8, and even 16 socket systems are becoming commonplace with as many as 160 physical cores available. Compared to the flat memory model and single processors that most operating systems were designed on initially, the landscape of performance and locality has become far more lumpy. Unfortunately, while applications and programming languages have been quick to adopt features that allow programmers to take advantage of parallelism with shared coherent memory, not all applications and particularly operating systems have been fast to develop implementations that can take advantage of all these processors. Beyond simply multithreading OS components to try and deserialize common routines, current state of the art is simply to try and allocate memory from your local NUMA node, rather than randomly within the physical address space. Local node memory accesses can have up to a 4-fold latency advantage over remote memory accesses in machines such as the Westmere-EX architecture that have noncrossbar interconnect topologies (as shown in Figure 6.1).

Maintaining node local memory accesses allows applications to scale performance across processors within a system while still maintaining the convenience of hardware shared memory semantics. Unfortunately, for applications where shared memory is in use the cost of cache coherence goes up substantially on high NUMA factor machines when even the smallest amount of memory is shared between execution on different sockets. Even today in NUMA-aware operating systems, there is substantial sharing that must take place as the OS implicitly manages much of the synchronization on behalf of the application. This synchronization allows a coherent view of process scheduling, memory management, and device access which is multiplexed to multiple

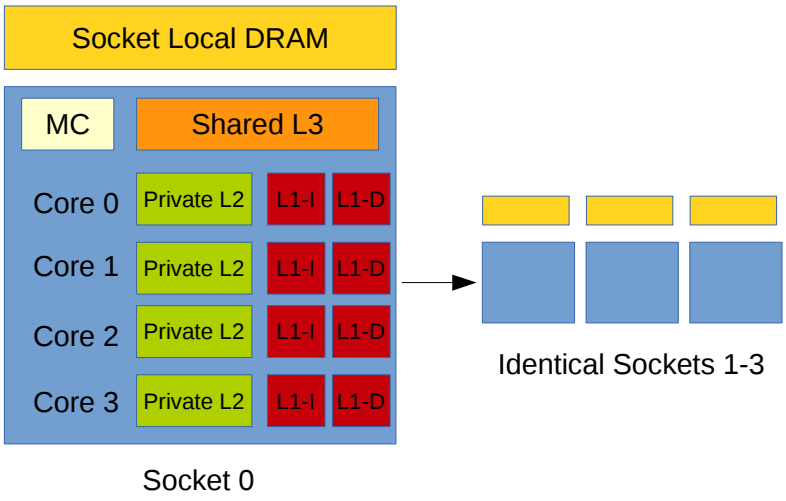

Figure 6.1. Typical cache layout of multisocket machines circa 2013.

users.

For many-threaded user applications there is a body of work on developing tiling systems for large data that help to ensure that only threads within the local socket are typically accessing the majority of the socket-local memory. Sharing of memory, which results in coherence operations, typically only occurs at the edge of these tiling boundaries. Within the operating system, however, a single routine or system will often be executed on all processors regardless of where the data structure allocation initially occurred. This means that the majority of memory accesses may be to a remote node, and closely spaced or simultaneous calls into the OS result in heavy locking and cache coherence operations.

Our previous observation that I/O system calls (pread, pwrite) generate large amounts of operating system execution in our webserver, database, and middleware applications, limiting absolute performance as measured by looking at application throughput per unit time. Throughput for many of these applications can be improved both by improving single threaded performance, which we examined in previous chapters, or by improving the scalability of performance across many multiple executing threads. To demonstrate the multicore scalability problems within the OS, we first examine in detail the Linux block I/O subsystem design as an example of how legacy OS designs are not scaling well into the many-core era. We then look at a similar, but constrained, problem of speeding up a storage device driver that is hampered

by the same problem as the block I/O subsystem to determine the improvement we can achieve by rearchitecting the OS data structures and routines to be cache and coherence topology aware. Our cache-aware I/O handling improvements require absolutely no modifications to hardware, as evidenced by the fact they were implemented on real existing hardware. We show that we can improve the absolute throughput of the I/O handling system by as much as a factor of 5. As a result, unmodified enterprise applications that are OS and I/O heavy see performance benefits up to 160%, demonstrating the very real benefit of OS and hardware co-design.

6.1 OS Performance Scaling for I/O

As an example of one such subsystem that is under heavy contention from multiple user threads, we look at the functionality provided by the Linux block layer. From the application perspective doing I/O may be as simple as doing *pread()* and *pwrite()*, but within the operating system there are numerous subsystems this request traverses before the I/O is issued and then completed back to the application. Figure 6.2 provides a functional view of the Linux block layer to show the various logic steps that occur within the OS while handling I/O. For simplicity we do not include the filesystem or page cache layers which introduce additional complexity including buffer copying and deferred I/O alignment, read ahead, and issue.

6.1.1 I/O Subsystem Architecture

Figure 6.2 illustrates the architecture of the current Linux block layer. Applications submit I/O via a library, libaio for asynchronous I/Os, as a data structure, called a block I/O. Each block I/O contains information such as I/O address, I/O size, I/O modality (read or write), or I/O type (synchronous/ asynchronous). [1] Once an I/O request is submitted, the corresponding Block I/O is buffered in the staging area, which is implemented as a queue, denoted the *request queue*.

Once a request is in the staging area, the block layer may perform I/O scheduling and adjust accounting information before scheduling I/O submissions to the appropriate storage device driver. Note that the Linux block layer supports pluggable I/O

[1] See *include/linux/blk_types.h* in the Linux kernel (kernel.org) for a complete description of the Block I/O data structure.

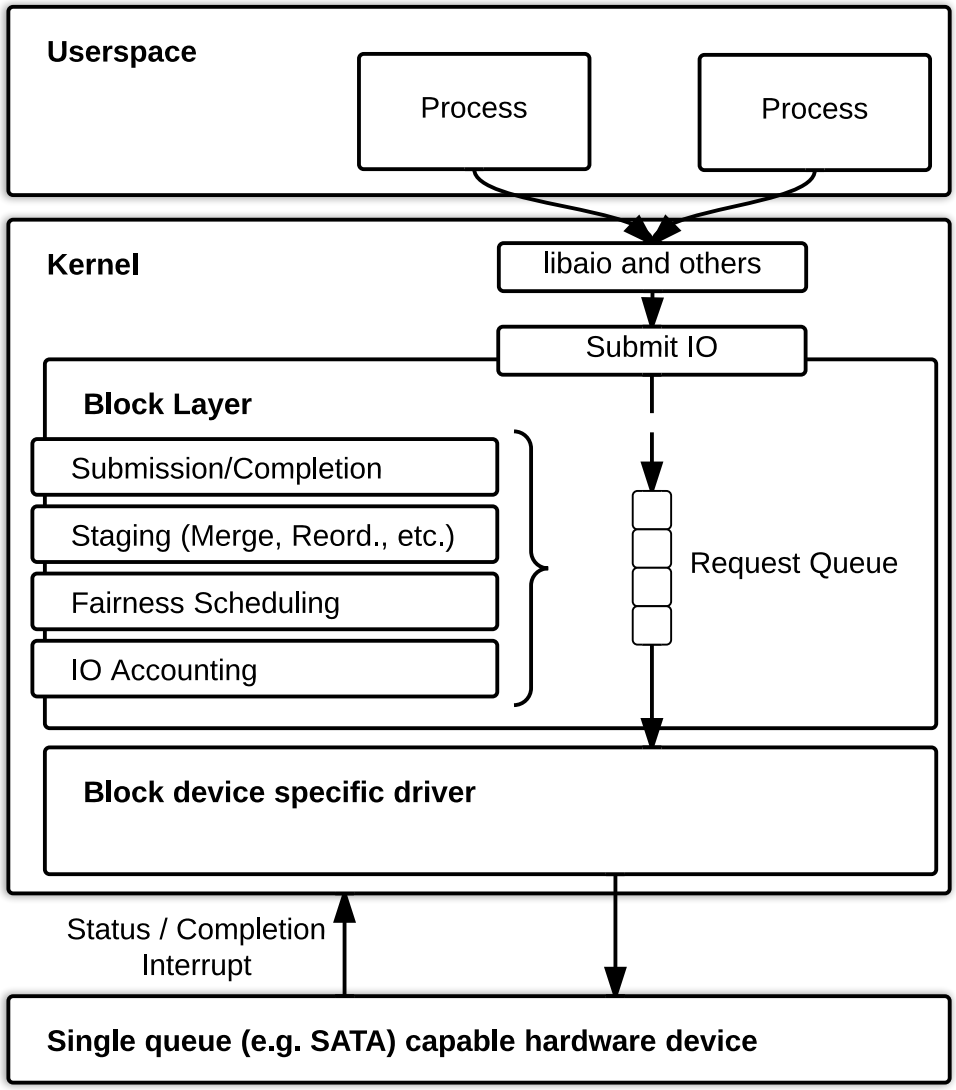

Figure 6.2. Current Linux 3.2.0 block layer design.

schedulers: noop (no scheduling), deadline-based scheduling, and CFQ that can all operate on I/O within this staging area. The block layer also provides a mechanism for dealing with I/O completions: each time an I/O completes within the device driver, this driver calls up the stack to the generic completion function in the block layer. In turn the block layer then calls up to an I/O completion function in the libaio library or returns from the synchronous read or write system calls, which provides the I/O completion signal to the application.

The single point of control flow for a device within the current block layer is the request queue structure. One such queue is instantiated per block device. Access is

uniform across all block devices and an application need not know what the control flow pattern is within the block layer. A consequence of this single queue per device design, however, is that the block layer cannot support I/O scheduling across devices.

Out of this design we identify the three major features that a block layer implementation must retain regardless of implementation:

- **Single Device Fairness** Many application processes may use the same device. It is important to enforce that a single process should not be able to starve all others. This is a task for the block layer. Traditionally, techniques such as Completely Fair Queuing (CFQ) [86] or deadline scheduling [87] have been used to enforce fairness in the block layer. Without a centralized arbiter of device access, applications must either coordinate among themselves for fairness or rely on the fairness policies implemented in device drivers (which rarely exist).
- **Single and Multiple Device Accounting** The block layer should make it easy for system administrators to debug or simply monitor accesses to storage devices. Having a uniform interface for system performance monitoring and accounting enables applications and other operating system components to make intelligent decisions about application scheduling, load balancing, and performance. If these were maintained directly by device drivers, it would be nearly impossible to enforce the convenience of consistency application writers have become accustomed to.
- **Single Device I/O Staging Area** To improve performance and enforce fairness, the block layer must be able to perform some form of I/O scheduling. To do this, the block layer requires a staging area where I/Os may be buffered before they are sent down into the device driver. Using a staging area, the block layer can reorder I/Os, typically to promote sequential accesses over random ones, or it can group I/Os to submit larger I/Os to the underlying device. In addition, the staging area allows the block layer to adjust its submission rate for quality of service or due to device back-pressure indicating the OS should not send down additional I/O or risk overflowing the device's buffering capability.

The I/O subsystem of the operating system is not nearly as simple as it might seem to a naive application writer. The multiplexing, fairness, and performance enhancements

that the operating system performance as a result of a single system call are the reason that large fractions of total instruction counts for an application workload are executed within the operating system for workloads such as databases, webservers, memcached, and middleware suites. Performing all these operations also introduces significant accounting overhead, which in turn requires locking around data structures that may be accessed on multiple CPUs in addition to having to perform cache to cache transfers of the actual data structure for access.

6.1.2 Performance Scalability Issues

To examine how the OS I/O subsystem scales up from a single thread to many threads we analyzed the structure and performance of the block layer on several common HPC platforms equipped with high-factor NUMA multicore processors and high IOPS NAND-flash SSDs. We found that the block layer was introducing considerable overhead for each I/O; specifically, we identified three main problems, illustrated in Figure 6.3.

1. *Request Queue Locking*: The block layer fundamentally synchronizes shared accesses to an exclusive resource: the I/O request queue. (i) Whenever a block I/O is inserted or removed from the request queue, this lock must be acquired. (ii) Whenever the request queue is manipulated via I/O submission, this lock must be acquired. (iii) As I/Os are submitted, the block layer proceeds to optimizations such as plugging (letting I/Os accumulate before issuing them to hardware to improve cache efficiency), (iv) I/O reordering, and (v) fairness scheduling. Before any of these operations can proceed, the request queue lock must be acquired. This is a major source of contention.
2. *Hardware Interrupts*: The high number of IOPS causes a proportionally high number of interrupts. Most of today's storage devices are designed such that one core (core 0 on Figure 6.3) is responsible for handling all hardware interrupts and forwarding them to other cores as soft interrupts regardless of the CPU issuing and completing the I/O. As a result, a single core may spend considerable time in handling these interrupts, context switching, and polluting L1 and L2 caches that applications could rely on for data locality [88]. The other cores (core N on Figure 6.3) then also must take an IPI to perform the I/O completion

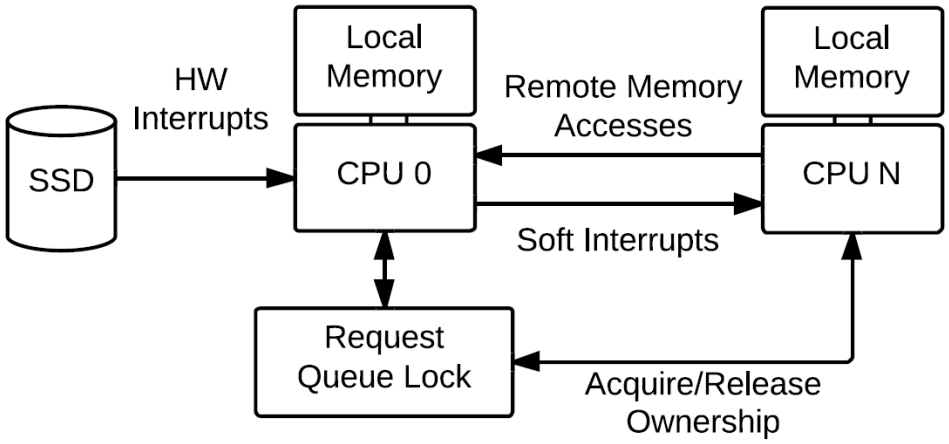

Figure 6.3. Simplified overview of bottlenecks in the Linux block layer.

routine. As a result, in many cases two interrupts (costing 1–2 microseconds each depending on machine architecture [80], and context switches are required to complete just a single I/O.

3. *Remote Memory Accesses*: Request queue lock contention is exacerbated when it forces remote memory accesses across CPU cores (or across sockets in a NUMA architecture). Such remote memory accesses are needed whenever an I/O completes on a different core from the one on which it was issued. In such cases, acquiring a lock on the request queue to remove the block I/O from the request queue incurs a remote memory access to the lock state stored in the cache of the core where that lock was last acquired, the cache line is then marked shared on both cores. When updated, the copy is explicitly invalidated from the remote cache. If more than one core is actively issuing I/O and thus competing for this lock, then the cache line associated with this lock is continuously bounced between those cores.

6.1.3 Performance Scalability Results and Observations

Figure 6.4 shows IOPS throughput as a function of the number of CPUs that are submitting and completing I/O to a single device simultaneously. We observe that when the number of processes is lower than the number cores on a single socket (i.e., 6), throughput improves or is at least maintained as multiple CPUs issue I/O. For 2,

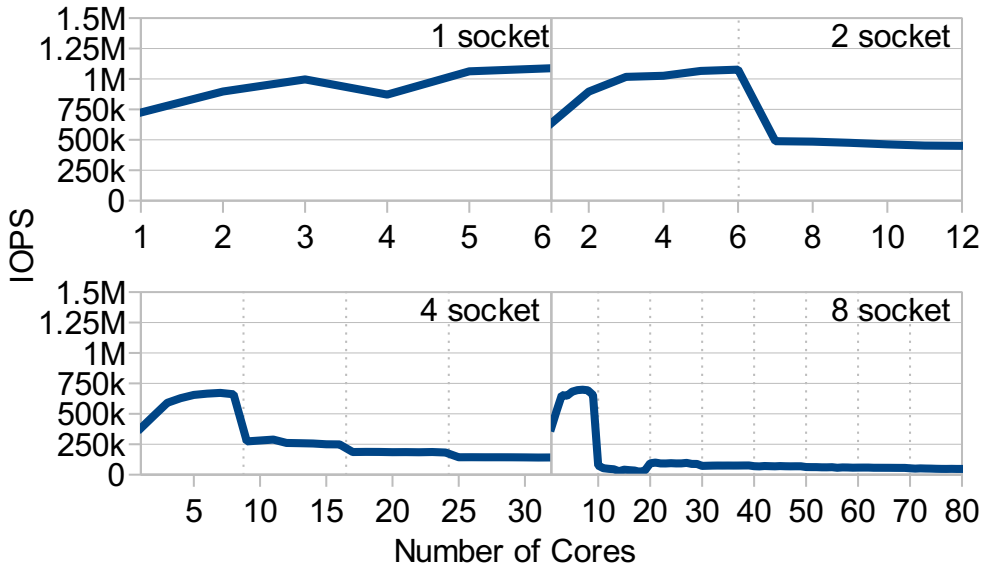

Figure 6.4. IOPS throughput of Linux block layer as a function of number of CPU's issuing I/O. Divided into 1, 2, 4, and 8 socket systems. (Dotted line show socket divisions.)

4, and 8-socket architectures which have largely supplanted single socket machines in the HPC space, when I/O is issued from a CPU that is located on a remote socket (and typically NUMA node), absolute performance drops substantially regardless of the absolute number of sockets in the system.

We attribute this to the fact that remote cacheline invalidation of the request queue lock is significantly more costly on complex 4 and 8 socket systems where the NUMA-factor is high and large cache directory structures are expensive to access. On 4 and eight 8 architectures, the request queue lock contention is so high that multiple sockets issuing I/Os reduces the throughput of the Linux block layer to just about 125k IOPS even though there have been high end solid state devices on the market for several years able to achieve higher IOPS than this. The scalability of the Linux block layer is not an issue that we might encounter in the future, it is a significant problem being faced by HPC in practice today.

Performance scalability that works on a single socket likely reflects the fact that most OS subsystems have not been substantially modified since they were designed to work on single processors or SMP systems where the cost of coherence was relatively low. We were shocked to see that not only does performance not scale up in multipro-

cessor systems, but that it actually was reduced by as much as a factor of 10× in these multisocket systems. This provides great opportunity for improvement, however, through operating system design, to simply match the current state of hardware design, which has obviously evolved more rapidly than OS design and implementation.

By evaluating an existing production OS subsystem for architectural interaction we learned two key things. Most importantly, looking at OS and architectural co-design issues on a single socket system is unlikely to show the true magnitude of the problem. This is due to the much higher relative cost of cache coherence on multisocket machines. As a result, we believe the results obtained via simulation with our previously proposed solutions are likely to be substantially understated. We also observe that scalability of a solution is a more important a factor of performance than any particular data point. Designing systems that perform well on both fast single socket systems as well as on multisocket systems is difficult because in isolation the best design may be vastly different on two systems. For example, designing cache friendly operating system components may need to be architecture aware and have different behavior when executing on a system with shared L3 caches within a socket, or having only private caches. When off-chip eDRAM caches become commonplace, as proposed in the upcoming IBM Power 8 processor, the cost model for locality awareness may change again, and the OS needs to be intelligent enough to take advantage of this.

Redesigning the Linux block I/O subsystem to fix these problems is a major undertaking that will likely take many years to complete due to the required backwards compatibility that must be maintained in the API both up and down the stack. To evaluate how much better an intelligent OS co-design might be we chose to reimplement a smaller OS subsystem that exhibits similar scaling properties. Specifically we look at the device driver for a DRAM backed storage device, which is meant to model a future high IOPS device based on technology that would have a device latency that is less than 1 microsecond, thus placing the OS driver implementation on the critical path for absolute performance.

6.2 A Cache Aware Storage Device Driver

In the previous sections we have shown that I/O, and thus operating system performance, is on the critical path for application performance in many enterprise applications. One technology that has dramatically increased the need for OS I/O performance to improve is NAND-flash. Its unique combination of price (10× less than DRAM but 10x higher than magnetic disk), latency (100× higher than DRAM, but 100× faster than magnetic disk), and power consumption (10× lower than DRAM, when viewed as a holistic device including required periphery). The NAND-flash market is growing so quickly that in 2013 it is expected to surpass the DRAM market according to research firm IDC. Compared to magnetic disk, raw flash device read latencies are just 30–50 microseconds, and when combined with external (powercut safe) write combining, the write latency can be just several microseconds. As technology progresses on alternative memory technologies, NAND flash is likely to be supplanted by a yet better technology for storage devices. Phase change (PCM) and spin torque magnetic RAM (STT-RAM) are two leading technologies that provide nonvolatility combined with nanosecond class read and write access. While both these technologies have significant work necessary to improve their density to be competitive with NAND-flash, their 100× latency improvement over NAND-flash makes them attractive for high performance applications even at lower capacities and higher costs.

These future technologies come with a variety of unique properties that make designing a storage device around them hard. They typically have 10–100× longer write latencies and limited write endurance. To account for this, most major storage vendors believe that some form of a log-based filesystem must be implemented within the device such that in-place writes are not required, and back block management can be handled transparently to higher levels in the operating system. So while these technologies are an improvement over NAND-flash, they are not a panacea, and the fundamental log-based nature of these drivers will remain in tact. To examine the impact that a complicated driver has on performance we examine the fraction of execution time that an enterprise application such at TPC-C spends executing within a device driver for a log based filesystem such as the Fusion-io ioDrive2 NAND-flash

based device.

Table 6.1 shows the effect on performance for the TPC-C suite running on top of mysql when we control the device latency without modifying the device driver or application stack. This was done by using a prototype DRAM backed device so that the device latency could be controlled from 500ns to 1 millisecond. What we observe is that when moving from 1 millisecond to 100 microseconds, there is a dramatic improvement in application throughput, even though the amount of execution time spent within the device driver nearly doubled. As the technology latency continues to be reduced, however, the relative amount of time spent within the device driver for the device goes up dramatically because it is essentially a fixed cost per I/O, and the total number of I/Os that can be done per fixed time unit is increasing due to the smaller technology latency. Because of the increasing fraction of execution that is spent within the device driver there is now relatively less time available for computation within the TPC-C benchmark suite itself, which is why performance does not continue to ramp as the device latency continues to decline. These data show that in the future an OS device driver can soon be on the critical path of device and thus application performance. Simply put, Amdahls law appears to have struck, with both the application and underlying storage technology improving, but the OS implementation is not scaling accordingly

6.2.1 Performance Breakdown of Device Driver

To understand why the fraction of execution spent within the device driver consumes such a large portion of the execution time, we look at the structure of the typical storage device driver. Figure 6.5 shows the process of our device driver when handling I/O completions from the device. First, the interrupt is taken, which shifts control of the processor into the device driver. Then within the interrupt handler one or more registers are read from the device to indicate the completion record that the device has just completed. After determining which I/O has completed, the driver then updates its data structures for outstanding I/O and any mapping information it is maintaining. After it has updated its internal state such that the data are now ready for use by the application, it asynchronously signals the completion to the

Table 6.1. Performance of TPC-C and device driver utilization for varying storage device latencies.

Technology Latency	% Exec. In Device Driver	TPC-C Throughput
1ms	2.3%	1.00
100us	15.3%	3.21
50us	22.8%	3.72
10us	30.9%	3.85
1us	44.2%	4.11

application, indicating that the I/O has completed. In this diagram we have omitted the operating system block layer, which actually sits in between the device driver and the application and performs yet more accounting on behalf of the OS as described in the previous section. After signaling the application that the I/O is ready, the driver then returns down its call stack and sends a completion acknowledgment to the device indicating that it has serviced this request. The interrupt then has completed, and CPU control is then transferred back to the application running on the CPU. This process should look very similar to Figure 6.2 as the device drivers are actually doing the interrupt handling and I/O completions for the Linux block layer.

One important aspect worth noting in Figure 6.5 is that this entire process actually happens within just a single CPU on the system. While this provides good cache locality on CPU 0 for the operating system, it also limits the speed at which I/O can be completed to the single threaded performance of one CPU in a system. Figure 6.6 shows the fraction of the CPU utilized while increasing the I/O rate performed by the application for a device that has 1us access time and should theoretically be able to do 1 million I/Os per second. We observe that a single-threaded device driver is not sufficient to provide the throughput required for this device and its performance is limited to just over 100k IOPS due to being completely CPU bound. This supports the observation in Table 6.1 that application throughput does not scale up with faster device technology because the absolute performance of the device becomes limited by the OS/driver performance, which in turn limits application throughput. This 100k IOPS rate is far below the rate that the Linux block layer can handle (approximately 800k), which means that the device driver, not the block layer is

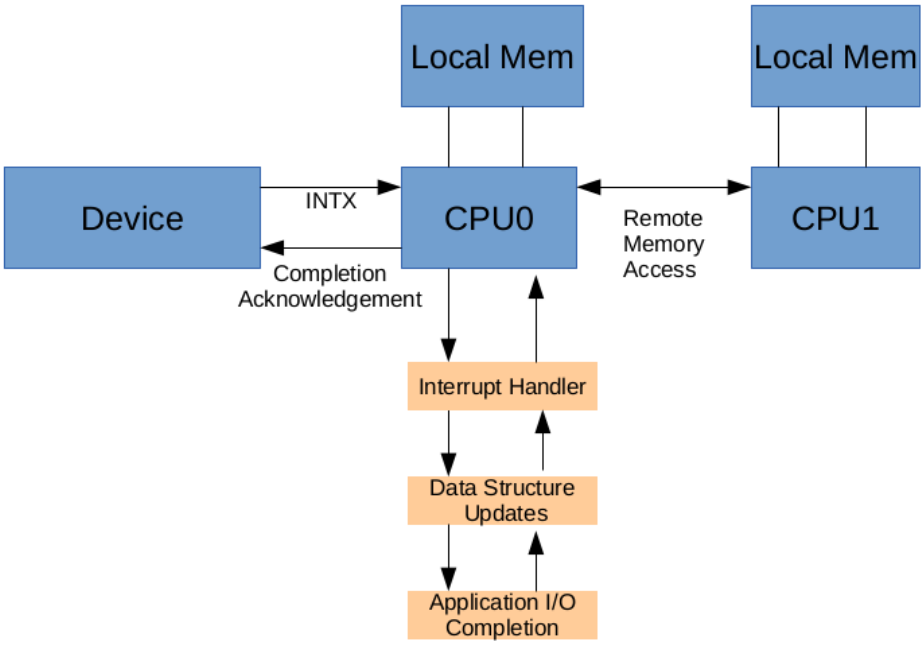

Figure 6.5. Device driver handling of I/O completions.

currently the bottleneck that needs improvement. We also examined the scalability of this implementation across processors issuing I/O, as we did for the linux block layer, and found a nearly identical trend. Maximum performance was achieved when issuing I/O from a single socket within a system and as soon as we began issuing I/O from alternate sockets, signaling the I/O completion to the remote sockets caused a reduction in performance due to the additional IPI required.

To determine the limiting factor within the device driver, we instrumented the driver to provide two short circuit paths that helped identify the bottlenecks within the driver. The first path eliminated the data structure and application I/O completion from the driver, it simply acknowledged the completion to the device without performing any accounting. With this modification we found that the device driver was able to perform just over 1 million I/Os per second. Though there were no actual I/Os being returned to the application, this ruled out the actual interrupt handling and return as the limiting factor to performance. The second short circuit path within the driver handled the interrupt and updated the in-driver data structures, but did not acknowledged the I/O to the application. With this modification, the device

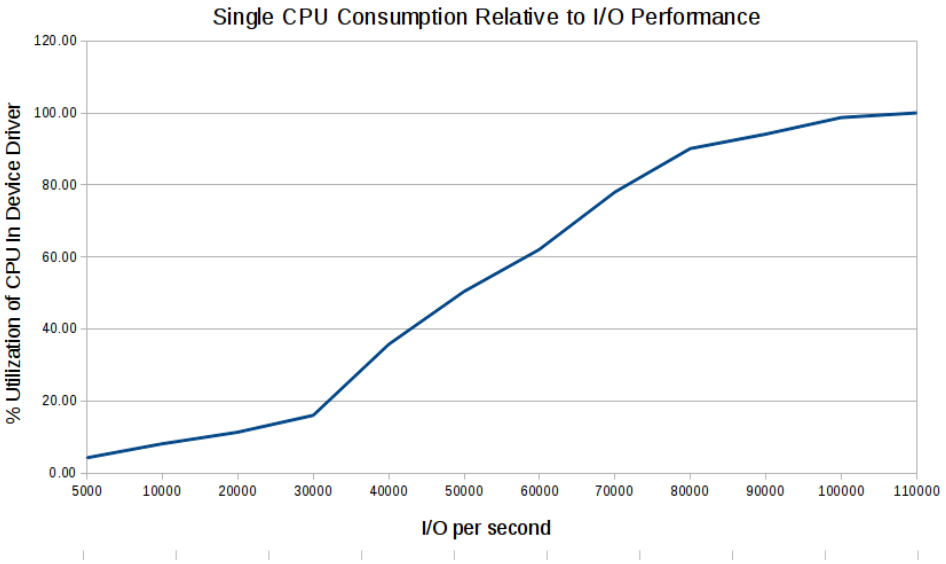

Figure 6.6. CPU utilization required to maintain a fixed I/O rate throttled at the application level.

driver was able to sustain slightly higher I/O rates of 164,000, compared to 110,000 in full operation. This gives us good insight that the internal data structure updates of the driver are by far the most costly operation, which intuitively make sense since these are also the most data intensive operations with a working set that is over 1MB and would not fit within the processor's L1 cache for all systems we examined in Table 6.2. While these internal updates do not account for all of the processing time, the addition of the OS completions only accounts for a further reduction of 50k IOPS compared to 850k.

6.2.2 Improving I/O Performance Via Multithreading

To improve the performance of this I/O device driver so that it can make use of future memory technologies, we have shown that a single-threaded device driver is not sufficient due to the amount of work that must performed per I/O. For magnetic drives and even NAND-flash devices it may be sufficient, but for technologies like phase change or STT-RAM, which are expected to have latencies below 200 nanoseconds, there would be little to no benefit of using them if we can not improve our operating system implementation to handle these low latency and high IOPS technologies. In Section 6.2.1 we showed that existing interrupt handling alone can scale to over 1

Table 6.2. Architecture of evaluation systems.

Platform (Intel)	SandyBridge-E	Westmere-EP	Nehalem-EX	Westmere-EX
Processor	i7-3930K	X5690	X7560	E7-2870
Num. of Cores	6	12	32	80
Speed (Ghz)	3.2	3.46	2.66	2.4
L3 Cache (MB)	12	12	24	30
NUMA nodes	1	2	4	8
Total Memory (GB)	24	48	192	1024
Memory Speed	DDR3-1333	DDR3-800	DDR3-1066	DDR3-800

Million IOPS if it does not have to perform substantial work within the interrupt handler. The CPU intensive portion of the driver operation is updating the internal driver state and completing the I/O back to the application. We now propose that to improve performance this OS driver must be split so that multiple CPUs can concurrently be processing I/O from the device so that they can be returned to the application at a higher throughput than available today.

Figure 6.7 shows the proposed architecture for our new device driver. The I/O completion in this model starts with a single hardware interrupt, just like the original driver, but rather than moving into completing the data structure updates directly, the interrupt handler simply locks a deferred task queue and inserts the identifier for the completion event into the queue. This queue insertion is an extremely lightweight operation beyond the operation of locking the task queue itself. After inserting the task into the queue, the interrupt handler can immediately return a completion acknowledgment to the device allowing the device to move on to its next request. After returning the completion acknowledgement to the device, the interrupt handler must now send an interprocessor interrupt (IPI) to the CPU on which the device driver structure updates and OS I/O completion should complete. By issuing these IPIs round robin across additional processors on a given socket, the computationally intense portion of the I/O completion is now distributed across multiple processors. When the deferred task processor takes this interrupt, it obtains a lock on the deferred task queue ensuring mutual exclusion with other processors also performing I/O completions. Because the use of data within the deferred task queue is mutually exclusive (no two processors will be completing the same I/O due to round robin

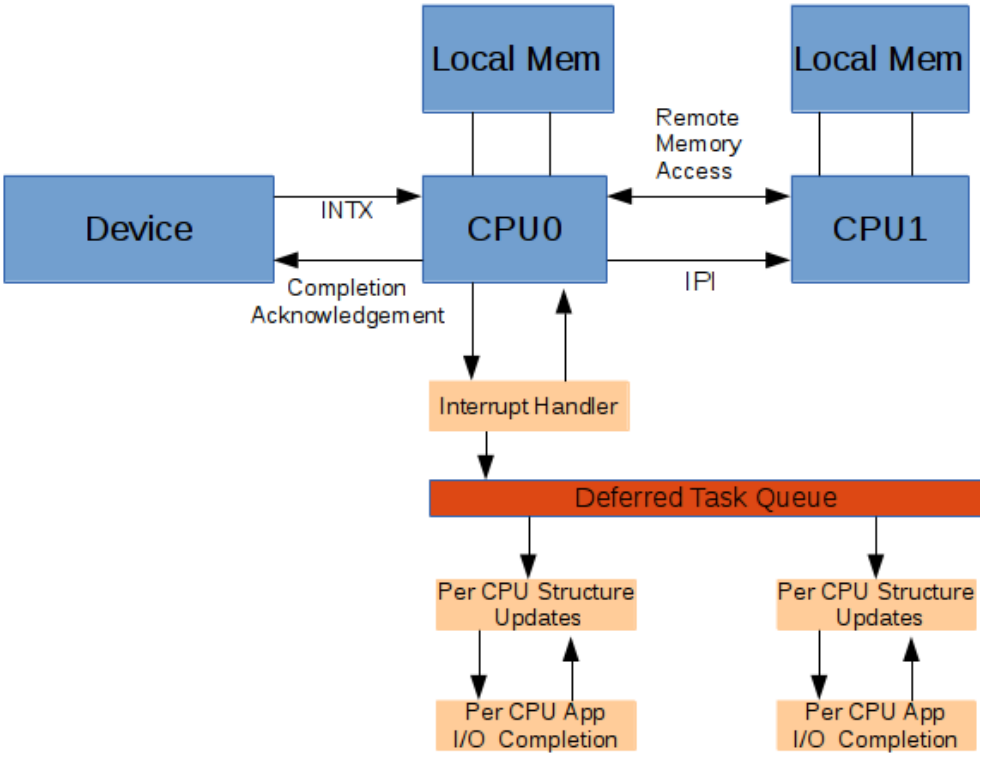

Figure 6.7. Multithreaded device driver architecture with distributed completions at expense of single threaded latency.

task distribution) the lock on the task queue must only be held for data structure correctness so that this circular buffer does not over/underflow. The lock need not be held to prevent access; thus a read/write lock can be used and only upgraded to a write lock when removing the deferred task. Because the interrupt handling portion of the device driver can perform over 1 million IOPS before saturating a single CPU as shown in Figure 6.6, leaving this portion serialized should not become a bottleneck to performance even in our multithreaded device driver design.

This new device driver architecture has the potential to scale up I/O performance, but it also introduces several issues that we need to be aware of during evaluation:

1. *InterProcessor Interrupts*: To enable multithreading of the I/O completions we must have a signaling mechanism to the remote cores to indicate that there is work for them to do. While those threads could certainly spin, busy waiting for work to arrive, this would effectively burn up all the additional CPU time on those processors, wasting power and performance at the application level. This leaves sending an IPI as the best mechanism for kicking them to initiate

performing the device driver work. Unfortunately, this means that for any single I/O we have now moved from having one interrupt to two (though distributed across cores). This has the negative effect of increasing I/O latency as well as interrupting more CPUs, which would otherwise be executing application code.

2. *Decreased Cache Locality*: As shown in previous chapters, both operating system and user performance is affected by the cache locality they are able to achieve within a given CPU. By multithreading our OS routine, we are effectively decreasing cache locality by distributing it across multiple CPUs. This is likely to negatively affect the performance of the device driver routine in addition to negatively affecting the performance of user applications running on those CPUs.
3. *Increased Processor Synchronization*: When multithreading this OS routine, we have effectively introduced a producer (interrupt thread) and consumer (I/O completion thread) relationship through a new shared data structure. In this particular case it is a single producer, multiconsumer relationship. As a result to maintain consistency in the shared data structure there is now locking that must be introduced around the structure that will induce additional cache coherency operations that may reduce performance.

6.3 Experimental Results

6.3.1 Methodology

To evaluate the effectiveness of our proposed changes to even a self contained OS subsystem such as a device driver simulation, results were not an option because there are few simulators capable of running both OS level traces and multisocket processor configurations with full memory system modeling. Instead, we chose to implement our experimental changes within the Fusion-io ioDrive2 device driver, which was the fastest I/O driver commercially available when this work began in 2010. No changes were required to the Linux 3.2 kernel that dynamically loaded the Fusion-driver and communicated through the standard block layer ABI with our modified driver. The total changes required in the driver were less than 10,000 lines of C code including comments and white space.

Table 6.2 shows the four machines we evaluated the modifications on, which range from a single socket to an 8-socket architecture with 80 physical cores. In all experiments hyperthreading was turned off to ease in results analysis. All machines were equipped with prototype hardware based on DRAM as the technology and tuned such that it was capable of providing 3.0GB/s random read and write bandwidth from the PCIe bus for storage and a minimum of 1 million IOPS with a raw device latency detuned to 1 microsecond. This bandwidth is equivalent to the aggregate sequential 4k block size bandwidth of thirty 15,000 rpm SAS magnetic hard drives or random block access of 16,000 15,000 RPM SAS drives. The PCIe device access latency to storage in these systems ranged from 10 microseconds in the single socket machine to 135 microseconds in the 8-socket Westemere-EX platform. The additional latency in the many socket systems is due to the complex pci subsystem that has several active bridges to allow fan-out to as many as 26 PCIe slots. For raw device experiments, the open source tool fio was used to drive the system and requests were made to the block device directly with no additional filesystem layered on top of the device.

For application level experiments all systems were configured to use the XFS filesystem in ordered data mode to ensure data integrity at the expense of performance, as is common in enterprise environments. All external drivers for benchmark runs were driven from an identical Sandybridge-E system via 10Gb Ethernet to perform these migrations on real systems based on the Ubuntu 10.04 LTS distribution. By running on real machines with no ABI modifications required to applications we were able to look at the relative performance improvements across 5 workloads, including specWeb, specJBB, and specCPU (as used in previous Chapters); we additionally were able to add TPC-C and TPC-E workloads which are representative of common OLTP ad OLAP running under mySQL 5.2 as the database backend.

6.3.2 Overheads and Improvement With Multithreading

Figure 6.6 demonstrated that our device performance was limited by the single threaded nature of our OS driver implementation. By moving to the multithreaded driver, we have introduced additional complexity through a two-phased I/O completion. To determine the impact this has on any given I/O performance, we examined

the single CPU scaling by replicating the same experiment in Figure 6.6 on our new multithreaded driver. For this experiment, rather than sending the interprocessor interrupts to additional processors, we simply sent them to our new completion handler located on the same CPU as the interrupt handler. Figure 6.8 shows that our new multithreaded capable scheme is less efficient per I/O than the previous single threaded driver. Across all IOPS rates it consistently consumes more CPU time for a given I/O-load, and the peak I/O rate is actually lower on a single CPU because the additional CPU overhead causes the CPU to be maxed out at a lower I/O rate than the single threaded driver.

While some additional overhead was expected from our multithreaded implementation, the goal was to improve the absolute I/O performance of the system, which was limited to 110k IOPS on the single threaded version. To determine if our implementation scales up, we performed a similar experiment to the previous one, but we allowed the CPU completions to be round robin across all CPU's within a single socket of our Westmere-EP machine, which contains six CPUs. The percent utilization of the CPU on Figure 6.9 can now go up to 600% as the device driver could possibly consume all cores on the socket rather than just one. The serialized driver performance and multithreaded but on a single CPU data are included so that per I/O efficiency can be seen easily. We observe that our multithreaded driver does in fact allow performance to scale up to nearly 5x the initial serialized performance. At low IOPS rates (below 40,000) the multithreaded, multi-CPU driver is significantly less efficient than either the single threaded driver, or the multithreaded driver operating on a single CPU.

In addition to the IPI overhead that was introduced by the multithreaded driver, scheduling round robin completions across all cores distributes data among multiple CPU caches, rather than centralizing it within a single one. We believe this is why at low IOPS the multithreaded multi-CPU driver is the least efficient of the three options. When we approach IOPS rates at which the single CPU versions are consuming 50% or greater of one CPU, however, the multi-CPU driver regains some efficiency, presumably because there is enough work such that the caches have enough data to remain warm. From 100k–400k there is near linear scaling of IOPS

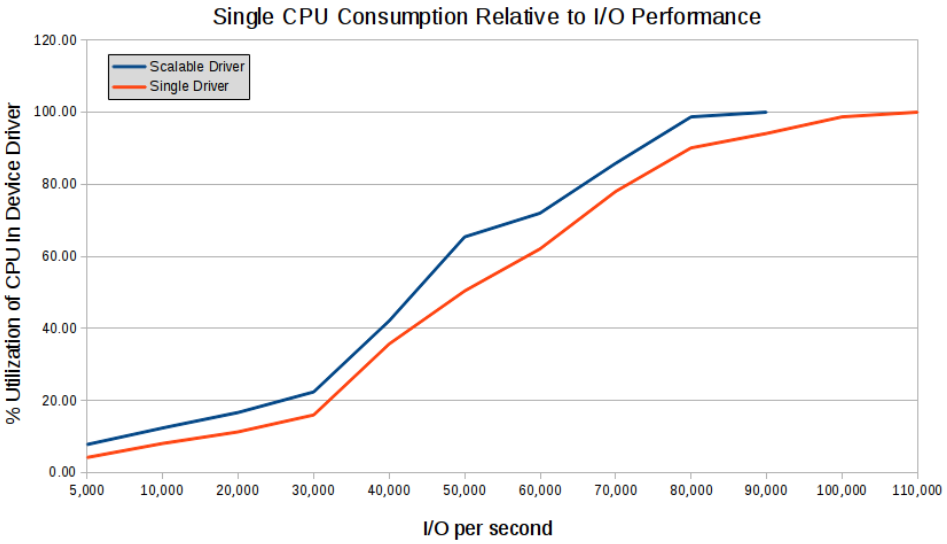

Figure 6.8. Multithreaded and single threaded driver efficiency on single CPU.

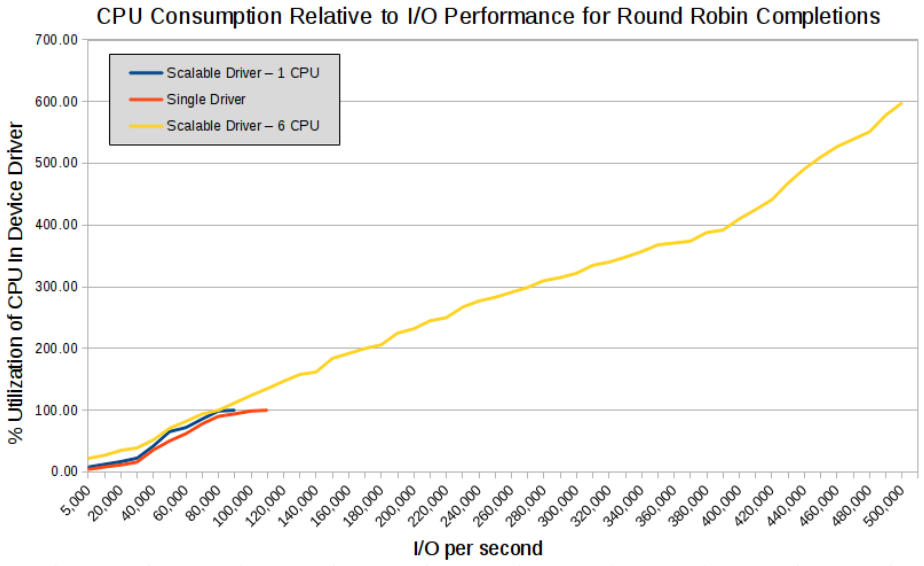

Figure 6.9. Multithreaded and single threaded driver efficiency on multiple CPUs.

with additional CPU usage, which is ideal, but above approximately 400% CPU utilization the IOPS rate does not continue to scale up. 400% processor utilization is approximately 80% of all six cores being utilized; this is very close to the point that the single threaded and multithreaded driver started leveling off. With the additional locking and cache contention that is introduced in the multithreaded multi-CPU version, it is not surprising that performance levels off before hitting 100% CPU utilization. While focusing on the areas for improvement we should not forget that the multi-CPU friendly version of the driver allows I/O performance on a CMP to scale up by a factor of 5. For applications limited by I/O performance, this should result in significant performance increases. Compared to the previous approaches of speeding up single threaded operating system execution, this approach takes a somewhat contrarian approach. We are willing to just maintain, or possibly even reduce, our single threaded performance and efficiency based on the belief that allowing performance scaling on a chip multiprocessor is actually more important for application throughput for I/O intensive application.

6.3.3 Multisocket performance

In the previous section we have shown that our multithreaded approach using additional interprocessor interrupts for signaling improves scalability at the expense of per I/O CPU efficiency. While this improves I/O performance on a single socket CMP, we have seen in the Linux block layer, that scaling I/O performance across multiple sockets can result in unexpected performance problems. To test how well our implementation performs in a large NUMA machine, we ran a similar experiment to that which we performed on the linux block layer. Specifically we looked at the total IOPS rate we could achieve when adding additional cores where the completions were sent back to the CPU on which the I/O had initiated from. This experiment was run on the Westmere-EX 8 socket system described in Table 6.2, which is known to have fairly expensive cache coherence, though it also has some of the largest on-chip caches which should help performance. Figure 6.10 shows the raw IOPS performance achievable when using additional cores to handle the completion, regardless of the CPU efficiency.

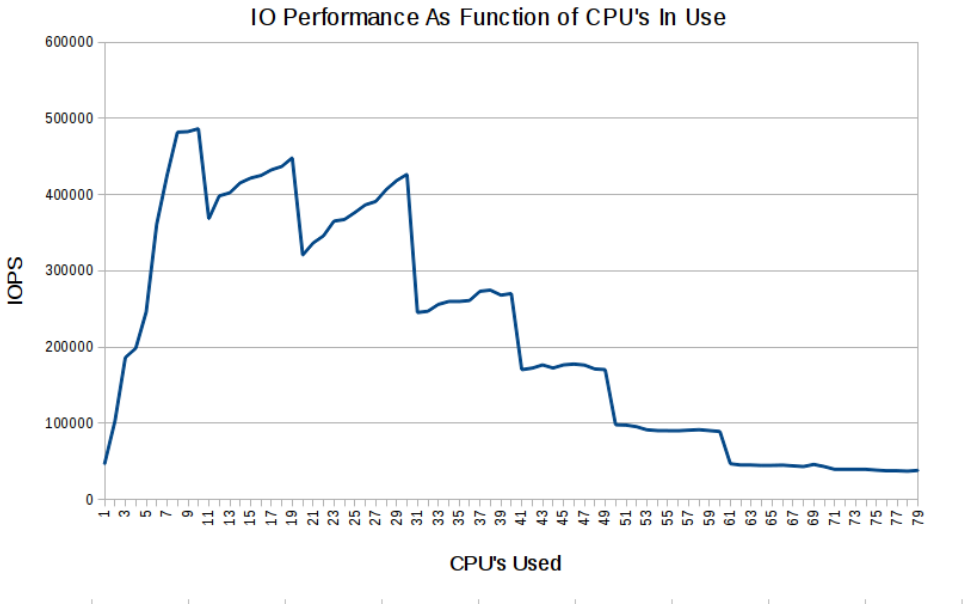

Figure 6.10. Performance of scalable driver on high NUMA-factor machine.

We observe that while I/O is occurring within the first 10 cores (which are on a single socket) we get very good scaling as we have seen previously. As soon as we introduce I/O from a second socket, however, there is a substantial reduction in total throughput. This throughput builds again as we use additional cores on the second socket, but it never exceeds the performance of the single socket only. As we introduce further sockets/CPUs, we see a similar pattern of a large drop when I/O completions move onto a new socket, then gaining back some of the throughput as more cores on that socket are used. If you consider the amount of CPU consumed versus the throughput, it is quite obvious that peak throughput and minimal CPU are found by using all CPUs on a single socket. This maximizes cache efficiency while minimizing the intrasocket cache coherence, which is very expensive once you have to go off-socket to other portions of the distributed directory. It is quite obvious that using per-CPU completion handlers across all cores in the system is neither energy efficient, nor appropriate for increasing throughput.

In Figure 6.10 we used CPUs in linear ordering such that each socket was maxed out before adding an additional socket for handling the I/O completions. In an 80 CPU system, it is very unlikely that only 1 socket of the system would be issuing

I/O. Because I/O is likely to come more randomly across the system, we want to be able to handle those I/Os on each socket locally and efficiently. To do this we tried yet another completion mapping. In this mapping we enable all completions to occur on the first CPU of each socket that the I/O was issued from, and only increase the number of completion handlers if that CPU becomes fully utilized. To do this we instrument a new check in the interrupt handler that polls the average utilization of the first CPU in each socket periodically to know if this CPU is fully utilized. This mapping policy should allow us to distribute the I/O completions across sockets to where I/O is being issued from at each application, yet keep the completions on just a single CPU within the socket to achieve improved cache utilization and when running real applications, reduce the amount of OS/User interference.

Figure 6.11 shows the absolute IOPS achievable on the Westemere-EX platform under this cache and coherence optimized per-socket mapping scheme compared to the cache oblivious mapping still included for reference. The socket-centric mapping performs nearly in-line with the sequential mapping up through 10 CPUs in use, but then achieves improved performance, though at lower CPU efficiency, up to 40 CPUs before it quickly starts degrading back to the sequential mapping performance. We attribute this to the fact that there is limited enough coherence that with completions being handled socket local, each handler achieves better cache locality than in the sequential mapping where one socket's local traffic can result in unfairness on the locking, thus starving other sockets.

6.4 Application Level Performance

In the previous Chapters 4 and 5 we primarily focused on how we can speed up the OS portion of an application on a single processor through improved cache policies. These approaches were limited because large caches that are shared between the user and OS portions of the code have enough capacity to accommodate the working set of both with limited interference. In this chapter, we identified that legacy OS design for the I/O subsystem (including device drivers) is drastically reducing the achievable I/O performance on modern hardware. Table 6.3 shows the maximum I/O throughput achievable with the 4 OS driver variants implemented for this work. While it is clear

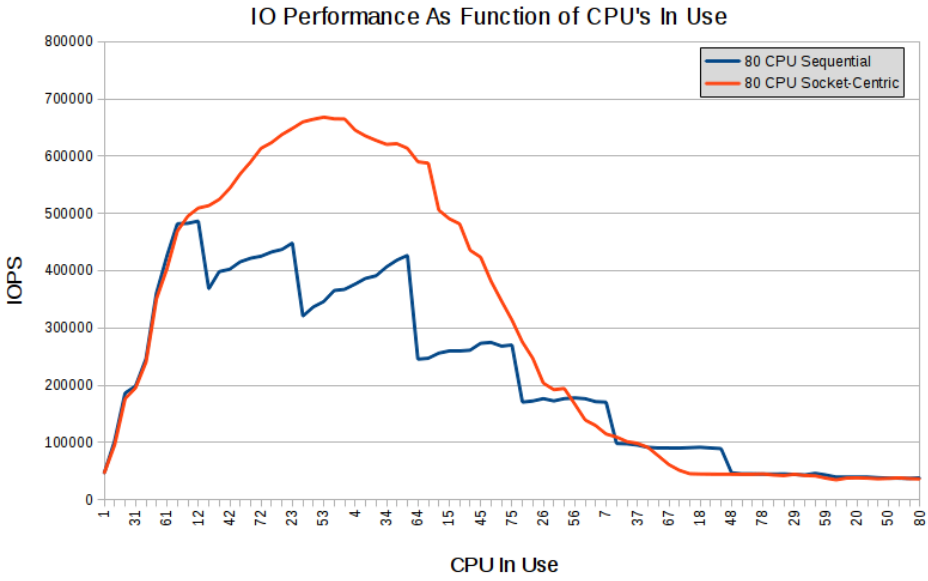

Figure 6.11. Performance of per socket completions on a high NUMA-factor machine.

that per I/O the single threaded driver is the most efficient in terms of CPU required, the multithreaded capable driver can perform nearly 5x the operations at near linear efficiency making it very likely a winner for applications for which I/O performance is critical. Finally our cache-aware multithreaded driver further improves upon the multithreaded driver by achieving the maximum performance possible but giving up yet more per I/O efficiency to achieve this. While energy efficiency is a key concern in modern computing systems, if an application is limited in overall throughput by one particular aspect of its execution, then spending more power to speed up that critical section is a good design decision.

In this section we evaluate the effect of three of our algorithmic variants on real work applications to see how our operating system improvements to match the capabilities of modern hardware and caching systems affects user throughput. On the Westmere-EX system described in Table 6.2 we execute 4 benchmarks – TPC-C, TPC-H, specJbb, and specWeb – across the single threaded (ST) device driver implementation, the multithreaded naive (MT-N), and multithreaded cache aware (MT-CA). All applications were run under identical hardware configurations for the duration of the benchmarks, which then self-report a throughput metric,

Table 6.3. I/O performance of various driver designs.

Driver Design	Max Throughput (IOPS)	CPUs Required
ST	115,000	1
MT Single Socket	490,000	6
MT Multi Socket - Naive	480,000	10
MT Multi Socket - Cache Aware	670,000	17

and performance is normalized to the single threaded implementation. This single threaded implementation is equivalent to state of the art in industry today having been based on the Fusion-io device driver implementation. We choose not to reexecute the specCPU workload used in prior chapters as they are not I/O intensive and no change in performance is expected as a result of our targeted changes. Problem sizes for all applications in this section were sized individually to maximize performance with the single threaded driver before the improved OS drivers were inserted.

6.4.1 SpecJBB 2005

Figure 6.12 shows the performance of specJBB2005 across our three driver versions. The multithreaded version of specJBB is able to achieve 12% improvement over the single threaded version with the cache-aware driver getting up to 20% throughput improvements. This improvement is less than the speedup we were able to achieve using cache partitioning techniques in Chapter 5. SpecJBB is representative of a middleware application that while OS intensive is not particularly I/O intensive. Thus even the speedup of 5x for I/O throughput translates into just 12% application speedup; using the system caches more effectively in the cache aware driver improves this by another 8% likely from better cache utilization of the driver, not further I/O throughput improvement. So while improved I/O is clearly not a panacea for all OS intensive applications, even those with some I/O can benefit from large throughput improvements and localized cache access of the OS routines.

6.4.2 SpecWeb2009

Figure 6.13 shows the performance of SpecWeb2009 across our three driver versions. The SpecWeb workload is an apache based webserving workload that is driven by an external request engine over the network. The apache webserver itself is heavily

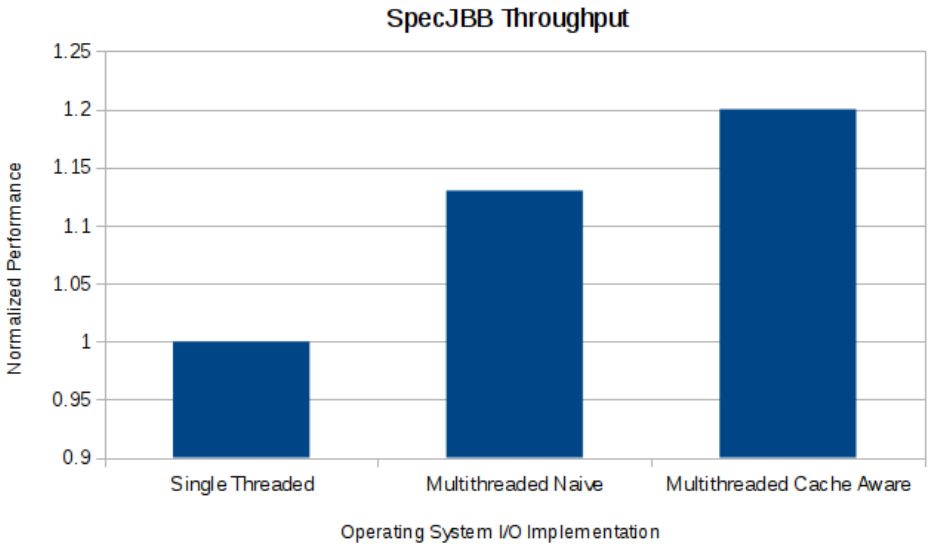

Figure 6.12. Performance of SpecJBB2005 on a high NUMA-factor machine.

multithreaded, and it is typical to run several processes per physical CPU to achieve maximum performance. When web requests are able to be serviced out of the DRAM cache maintained by Apache, requests are very fast, but those that miss in memory and must do I/O to be services are relatively long latency. Because you do not want concurrent requests blocked on these threads, oversubscribing the CPU with active threads allows those that are not blocked on I/O to continue. This heavy use of threading results on our 80 CPU test machine resulted in Apache using just over 600 threads for maximum performance, any of which could perform I/O.

Our multithreaded I/O driver is able to achieve a 30% performance improvement over the native driver thanks to the large increase in total I/O throughput. Further, the cache-aware driver is able to achieve a nearly 55% speedup, almost twice that of the naive implementation. Because of the massive number of threads and heavy use of the network stack (to send back results), having a smaller number of I/O completion threads results in improved cache utilization for the Apache threads that dominated execution on most cores on a socket while a single CPU was utilized for I/O and nearly 50% of an additional core was being utilized by the network driver.

We were a bit surprised at how well SpecWeb2009 responded to our cache-aware driver improvements given that the single CPU cache modifications in Chapter 5

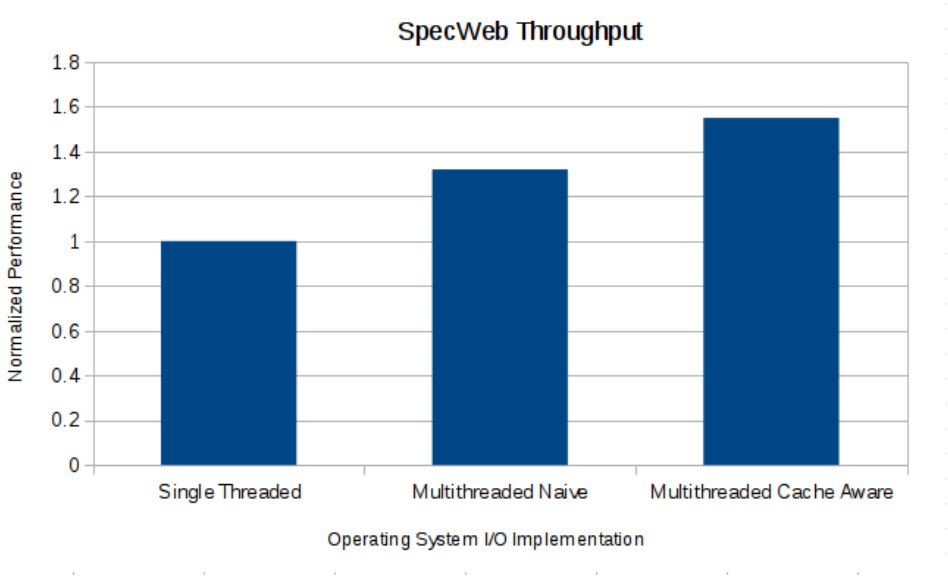

Figure 6.13. Performance of SpecWeb2009 on a high NUMA-factor machine.

did not show any improvements for our Apache workload. SpecWeb2009 reports throughput as its metric of performance, but average request latency is also available. The average request latency is more a function of the single thread performance than the total thread throughput and gives us an idea of how any individual request is being handled. We found that when moving from the single to multithreaded driver, the request latency of each individual apache request dropped by nearly 3x, yet with our work we had not sped up any individual I/O; the underlying technology remained the same. We then realized that much like a memory controller the I/O queuing delay was dominating the total time for the I/O to complete. Thus improving the throughput of the device driver had drastically reduced the queuing delay and improved the individual request latency.

In our cache segregation experiments, we were focused on speeding up an individual request by reducing the OS time required for any individual request. Because application performance is dominated by this aggregate queuing delay, speeding up any individual request makes little to no difference. Large improvements can come only from increasing the parallelism with which the OS can handle concurrent requests. This observation makes us believe that for server throughput-oriented applications, improving individual thread performance may simply be barking up

the proverbial *wrong tree*.

6.4.3 TPC-C

The TPC-C benchmark is an online transaction processing database suite for transactional online order query, entry, updates. To quote the tpc.org website,

> The benchmark is centered around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. While the benchmark portrays the activity of a wholesale supplier, TPC-C is not limited to the activity of any particular business segment, but, rather represents any industry that must manage, sell, or distribute a product or service.

This benchmark has long been one of the standards that large systems vendors such as IBM, Oracle, HP, and Dell compete ruthlessly to be at the top of standardized scoring system. A complex benchmark such as this is dependent on building a well-balanced system of CPU, memory, I/O resources, and even system interconnect because competitive machines for this benchmark typically have thousands of nodes. The end goal of these large clusters is to win contracts with companies such as Walmart, Amazon, or other high volume retails who are often looking for end-to-end solutions to their order managements and are willing to pay large amounts of money for a system that gives them a competitive advantage. In many ways, TPC-C is the gold standard for OS intensive benchmarking.

Unfortunately, in previous chapters the methodology used of simulation would not allow a large benchmark like TPC-C to warm up and execute realistically. We estimate that to run even a full TPC-C stack under the Simics execution environment used in Chapters 4 and 5 would have required nearly 4 years of simulation time. Because we have implemented our cache-aware OS I/O driver on real hardware, we are able to run the unmodified TPC-C benchmark on a machine that is representative of what a moderate size on-line retailer might actually run in production. Figure 6.14 shows that our multithreaded device driver implementation achieves nearly a 2x improvement in TPC throughput, thanks primarily due to the large improvement in IOPS achieved. Moving to the cache-aware version of the driver further improves

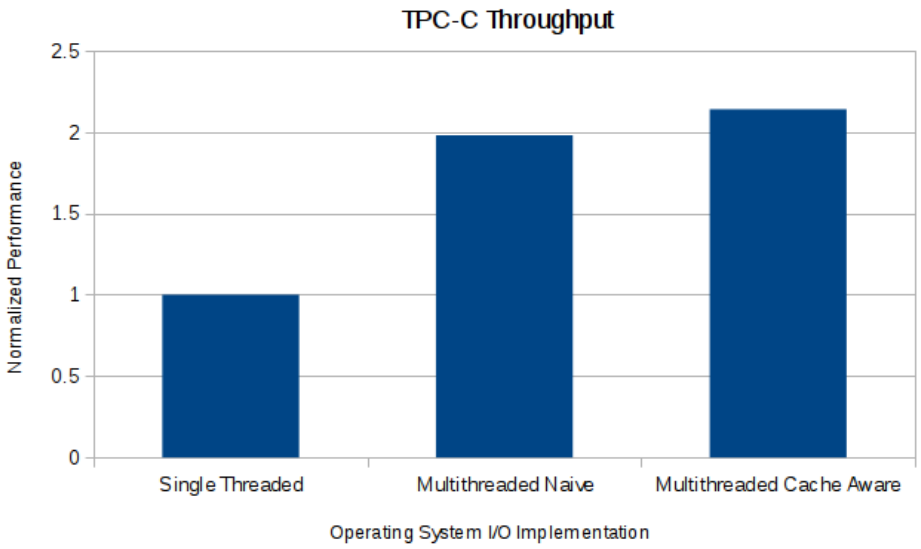

Figure 6.14. Performance of TPC-C on a high NUMA-factor machine.

performance but only by another 10%. During TPC-C execution we find that even with the cache aware driver, there are idle CPU resources available on the machine, indicating that the CPU heavy portion of the benchmark is still limited by the absolute performance of the I/O subsystem. Improving the efficiency at which those I/Os occur should be a secondary concern compared to further scaling up the performance of the I/O subsystem.

6.4.4 TPC-H

The TPC-H benchmark is an online analytic benchmark for decision support. From the tpc.org website,

> It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.

Business analytic and big data are a fast growing industry as retailers search for any relationship they may be able to exploit to increase profit margins. IBM, Oracle, and SAP sell entire hardware and software packages to try and meet these business

needs. Compared to TPC-C, TPC-H tends to be much more read-oriented in both main memory and from storage as looking for existing relationships such as high covariance between two data series is a common operation. The data set sizes for TPC-H also tend to be substantially smaller than those for TPC-C but are growing as retails look at larger datasets for yet more trends in customer habits to exploit for profit.

Similarly to TPC-C, running TPC-H in a full system simulation environment was not possible. Running TPC-H on our real hardware shows in Figure 6.15 that our naive multithreaded OS driver is able to achieve approximately a 70% speedup over the native single threaded driver. This is lower than the performance improvement seen in TPC-C because TPC-H performs 42.5% less total I/O query than TPC-C, and thus improvements in I/O throughput do not improve application throughput as much. When moving to the cache aware driver, TPC-H performance improved nearly another 70%, which was very surprising. We analyzed the TPC-H I/O behavior and discovered that transactional updates (that generated heavy I/O) were not regular during the TPC-H run; they tended to be batched, such that there was periodic behavior of reading in memory mostly, a period of write I/O traffic, read I/O traffic, then back to to the CPU and memory portion of computation. We believe that the cache aware I/O driver performed better than the naive version because it was more effective at preserving cache locality on 72 of the 80 processors rather than using all 80 for I/O periodically. We observe that the absolute I/O rate during the I/O heavy portions of the benchmark did not actually achieve the peak I/O rate of even the naive multithreaded driver. This means that the additional performance when using the cache-aware driver is not coming from additional I/O throughput; it is coming from decreased driver interference with the userspace benchmark processes. For TPC-H we conclude that improvements are coming from *both* improved OS I/O throughput and decreased OS interference with the application threads.

6.5 Conclusions and Future Improvements

In this chapter we have presented a different aspect of operating system and hardware co-design. Previous chapters have focused on architectural improvements

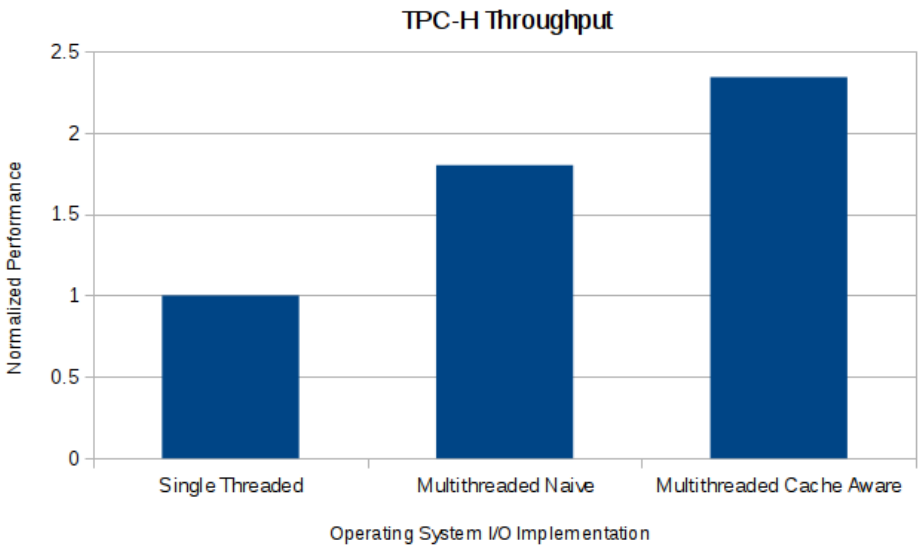

Figure 6.15. Performance of TPC-H on high NUMA-factor machine.

in the caching subsystem that can be applied with little or no operating system involvement. We found that for OS intensive applications, there was some benefit to segregating caches for applications in which the operating system execution was having a negative effect on user thread throughput. This exploration was based on the observation that cache locality is often the dominant factor in CPU performance when additional ILP is hard to extract from a thread of execution.

In this chapter we identify one aspect of OS performance, namely disk I/O, where current operating system implementations are not sufficient to maximize the performance of the non-CPU hardware (storage device). This leads to OS execution bottlenecking the I/O to the system, which in turn bottlenecks the overall application performance. We demonstrate that on a modern CMP device, an OS designer who is aware of the processor architecture can improve performance on existing storage devices by a factor of 5×. This throughput does not come without a slight trade-off in per I/O CPU efficiency, however, as additional interrupts and thread execution is required to capture this throughput. Overall this tradeoff is worthwhile, however, with our four OS intensive benchmarks showing between 10 and 160% speedup on real hardware. In addition to simply providing an architecture-aware driver design, we explore integrating the concept of cache-awareness in this multithreaded

driver, merging the concepts from Chapters 4 and 5 with the multithreaded driver design. We find that on average our cache-topology aware driver outperforms the naive multithreaded driver by 34% at the application level. This strongly supports our thesis that high performance systems must be operating system aware during execution to maximize the performance and energy efficiency of the system.

In addition to the work performed for this dissertation, additional observations have been made that will lead to additional related work. In this work, a storage device driver was improved such that it is approaching the limit of the generic block layer within the Linux operating system, as shown in Section 6.1.2. Suddenly, the full I/O subsystem design needs to be revisited or it will become the limiting factor to both application and device performance itself. Because of many of the observations presented in this chapter, a rewrite of the Linux block I/O subsystem has been initiated by the open source kernel maintainers, and an initial implementation has been merged into the mainline Linux kernel beginning in version 3.12.

As future memory technologies come online such as phase change memory and STT-RAM, the raw device latency will drop by an order of magnitude from where it is today utilizing NAND-flash. As part of the proposed solutions in this chapter we noted that the increased throughput achieved came at the expense of slightly more latency for any individual I/O due to the additional interprocessor interrupt signaling required. This latency overhead will grow more important as it dominates the total latency of the device and will need to be addressed. Possible solutions include i) dynamic threading, where interprocessor interrupts are used only when it is determined that the local interrupt handler is overloaded and it completes I/O in-line if possible. ii) MSI-X interrupt targeting, where devices are able to issue more than one interrupt and can send the interrupt selectively and directly to the CPU on which the I/O was initiated. This approach is where the Linux block I/O community is currently headed as the best long-term solution, though it does require device modifications to be compatible.

CHAPTER 7

DISCUSSION AND CONCLUSIONS

This chapter summarizes the topics explored in this dissertation and describes the impact it is likely to have on future architecture and operating system co-design.

7.1 Contributions

In this dissertation, we explore three techniques for improving the performance of industry standard operating system intensive workloads such as databases, webserving, and business to business middleware. We demonstrate that operating system performance has historically been ignored as an important part of designing large systems and even today there are large improvements to be had by taking a holistic view of performance and ensuring that both system software and hardware architecture work well together. To summarize the contributions, we briefly list the major highlights of each chapter in order:

- **Decomposing Operating System Performance.** In Chapter 2 we show that for system intensive workloads such as multimedia and webserving, the operating system contributes between 30 and 75% of the total instructions being executed. We then show that operating system execution has seen no speedup due to architectural improvements, only technology scaling, between the original 486 and the Pentium 4, one of the most aggressive single threaded performance processors every designed. This lack of OS performance scaling has resulted in large class of applications, that are dependant on OS performance, to have underperformed over the last 20 years compared to commonly benchmarked applications.
- **Offloading OS Application to Helper Cores.** In Chapter 4 we show that without operating system modification it is possible to utilize an operating

specific execution core to speed up OS execution without negatively affecting user thread performance. We provide a dynamic run-length predictor for OS execution, which predicts what sequences of OS execution are worth migrating to this additional processor versus those that are too short and will not show improvement due to the overhead of migration. We show that while performance can be improved via this technique, the energy efficiency of such an approach leaves much to be desired even with near perfect clock and power gating because the majority of the benifit is derived from having additional cache resources.

- **Segregating OS Execution in Local Caches.** In Chapter 5 we demonstrate that in multiway last level caches, it is possible to design cache policies that segregate OS and User cache references to decrease cyclical interference and improve performance. We look at permutations of cache policies including bank-partitioning, way-partitioning, and adaptive insert including segregation of only intstructions, data, or both. We find that while performance improvements are possible, large last level caches have enough capacity that making accurate segregation decisions is critical.
- **OS Co-Design Based on CPU Architecture.** In Chapter 6 we make the observation that many OS intensive applications are bottlenecked primarily on storage I/O. We look at the I/O performance of several modern architectures including large NUMA machines and observe that the operating system storage architecture is limiting I/O performance well below the performance achievable by modern hardware including NAND-Flash based drives. We decompose the performance problems of the I/O system to find that they exist primarily because driver and OS design has not been updated to match the needs of a modern machine architecture. We then propose a new architecture that more closely matches the processor and cache layout of current HPC architectures. Finally we show that this new architecture can improve raw I/O performance by over a factor of 5×, which improves application throughput by as much as 160%

7.2 Conclusions

When this work was begun, the initial focus was on improving the speed of a single thread of execution that alternated cyclically between user and operating system sections. Without additional ILP to extract through branch prediction or out of order execution, we turned towards caching as the primary place we could improve performance. While we were able to obtain moderate improvements, they were not so compelling as to be obvious solutions to both computer architects and operating system designers. In Chapter 6 we recognized that for operating system intensive applications, *performance* was often less about single threaded performance and more about aggregate system throughput. We identified the I/O subsystem of the operating system as a significant bottleneck to overall system throughput due to legacy design that was not optimized for today's complex multitiered processing and memory hierarchies. By redesigning just this one operating system subsystem we were able to improve overall application throughput by as much as 160%.

In computer architecture, improvements of 5 or 10% demonstrated through simulation are often viewed with skepticism. The fidelity of the simulation environment and assumptions made can play a large part in if the proposed idea is successful or not. It is unfortunate that the state of architectural simulation has not advanced to the point where we can run full system simulation of full workloads and operating systems easily, as this casts further doubt on the results of simulation based studies. In this work we were fortunate to be able to apply the concepts crafted via simulation to real OS implementation and production hardware system to not only match, but greatly exceed the performance expectations we had based on simulation.

While the OS I/O subsystem is just one small portion of the entire operating system, we believe the results from this dissertation are compelling enough that other additional OS subsystems will be redesigned for architectural awareness in the future. Our belief is supported by the fact that the concepts illuminated in this work have been applied to both a commercially shipping product and are currently being integrated into the Linux kernel by the open source community. As a result, we conclude that the initial thesis statement "This dissertation tests the hypothesis that intelligent use and co-design of operating system and hardware systems can improve

overall system performance and power efficiency for full system applications" is in fact true, and there are significant gains to be made by computer architects and operating system designers working more closely together in the future.

REFERENCES

[1] Intel Inc., *Intel Pentium 4 Processors 570/571, 560/561, 550/551, 540/541, 530/531 and 520/521 Supporting Hyper-Threading Technology.* [Online datasheet], Available: `http://www.intel.com`.

[2] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures," in *Proc. ISCA-27*, June 2000, pp. 248-259.

[3] D. W. Wall, "Limits of instruction-level parallelism," in *Proc. ASPLOS*, vol. 26, New York, NY: ACM Press, 1991, pp. 176–189.

[4] C. Hsu, U. Kremer, and M. Hsiao, "Compiler-directed dynamic frequency and voltage scaling," in *Workshop on Power-Aware Computer Systems*, 2000.

[5] C. Hsu and U. Kremer, "Dynamic voltage and frequency scaling for scientific applications," in *Proc. 14th Annu. Workshop Languages and Compilers for Parallel Computing (LCPC)*, 2001.

[6] S. Herbert and D. Marculescu, "Analysis of Dynamic Voltage/Frequency Scaling in Chip Multiprocessors," in *Proc. ISLPED*, 2007.

[7] S. Rele, S. Pande, S. Onder, and R. Gupta, "Optimizing static power dissipation by functional units in superscalar processors," in *Computational Complexity*, 2002, pp. 261–275.

[8] J. Butts and G. Sohi, "A Static Power Model for Architects," in *Proc. MICRO*, 2000.

[9] M. Annavaram, "A case for guarded power gating for multi-core processors," in *Proc. HPCA*, 2011, pp. 291–300.

[10] P. Bose, A. Buyuktosunoglu, J. A. Darringer, M. S. Gupta, M. B. Healy, H. Jacobson, I. Nair, J. A. Rivers, J. Shin, and A. Vega, "Power management of multi-core chips: Challenges and pitfalls," in *Proc. DATE*, 2012, pp. 977–982.

[11] Intel Inc., *Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge.* [Online], Available: `http://www.intel.com`.

[12] Y. Wang, S. Roy, and N. Ranganathan, "Run-time power-gating in caches of GPUs for leakage energy savings," in *Proc. DATE*, 2012, pp. 300–303.

[13] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. ISCA*, 1995, pp. 392–403.

[14] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang, "The case for a single-chip multiprocessor," in *Proc. ASPLOS*, 1996.

[15] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas, "Single ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proc. ISCA*, 2004.

[16] J. Redstone, S. J. Eggers, and H. M. Levy, "An analysis of operating system behavior on a simultaneous multithreaded architecture," in *Proc. ASPLOS*, 2000.

[17] D. Burger and T. Austin, "The simplescalar toolset, version 2.0." Univ. Wisconson-Madison Tech. Rep. TR-97-1342, 1997.

[18] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod, "Using the simos machine simulator to study complex computer systems," *Modeling Comp. Simulation*, 1997.

[19] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: a full system simulation platform," *IEEE Computer*, vol. 35(2), February 2002, pp, 55–58.

[20] L. Schaelicke, "L-rsim: A simulation environment for i/o intensive workloads," in *IEEE Workshop on Workload Characterization*, Los Alamitos, CA, 2000, pp. 83–89.

[21] M. Vachharajani, N. Vachharajani, D. Penry, J. Blome, and D. August, "The liberty simulation environment, version 1.0," *Perf. Evaluation Review: Special Issue on Tools for Arch. Research*, March 2004.

[22] J. B. Chen and B. N. Bershad, "The impact of operating system structure on memory system performance," in *Proc. SOSP*, 1993.

[23] A. Alameldeen, C. Mauer, M. Xu, P. Harper, M. Martin, D. Sorin, M. Hill, and D. Wood, "Evaluating non-deterministic multi-threaded commercial workloads," in *Proc. CAECW*, 2002.

[24] L. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese, "Impact of chip-level integration on performance of OLTP workloads," in *Proc. HPCA*, 2000.

[25] Windriver Inc., *Virtutech Simics Full System Simulator*. [Online], Available: `http://www.virtutech.com`.

[26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *Computer Architecture News*, vol. 39, no. 2, 2011.

[27] Byte Magazine, *Byte magazine's bytemark benchmark program*. [Online], Available: `http://www.byte.com/bmark/bdoc.htm`.

[28] R. Coker, *Bonnie++ filesystem benchmark*. [Online], Available: `http://freshmeat.net/projects/bonnie/`.

[29] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays," in *Proc. ISCA*, 2002.

[30] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-Effective Superscalar Processors," in *Proc. ISCA*, 1997.

[31] K. Yaghmour and M. Dagenais, "Measuring and characterizing system behavior using kernel-level event logging," in *Proc. USENIX*, 2000.

[32] T. Li, L. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio, "Understanding and improving operating system effects in control flow prediction," *Oper. Syst. Rev.*, 2002.

[33] N. Gloy, C. Young, J. B. Chen, and M. D. Smith, "An analysis of dynamic branch prediction schemes on system workloads," in *Proc. ISCA*, 1996.

[34] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska, "The interaction of architecture and operating system design," in *Proc. ASPLOS*, 1991.

[35] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache Performance of Operating System and Multiprogramming Workloads," *ACM Trans. Comput. Syst.*, vol. 6, no. 4, 1988, pp. 393–431.

[36] D. Nellans, R. Balasubramonian, and E. Brunvand, "A case for increased operating system support in chip multi-processors," in *Proc. IBM PACT2*, 2005.

[37] T. Li and L. K. John, "Operating System Power Minimization through Run-time Processor Resource Adaptation," *IEEE Microproc. Microsys.*, vol. 30, 2006, pp. 189–198.

[38] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: employing hardware migration to specialize CMP cores on-the-fly," in *Proc. ASPLOS*, 2006.

[39] J. A. Brown and D. M. Tullsen, "The shared-thread multiprocessor," in *Proc. ICS*, 2008.

[40] J. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar, "Using asymmetric single-ISA CMPs to save energy on operating systems," *IEEE Micro*, June 2008.

[41] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Proc. MICRO*, 2012.

[42] T. Sondag and H. Rajan, "Phase-based tuning for better utilization of performance-asymmetric multicore processors," in *Proc. CGO*, 2011.

[43] H. Stone, J. Turek, and J. Wolf, "Optimal partitioning of cache memory," in *IEEE Trans. Comput.*, 1992, pp. 1054–1068.

[44] L. Fand, F. Guo, Y. Solihin, S. Kim, and A. Eker, "Characterizing and modeling the behavior of context switch misses," in *Proc. PACT*, 2008.

[45] G. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," in *Proc. SC*, 2004.

[46] M. Qureshi and Y. Patt, "Utility-based cache partitioning: a Low-Overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO*, 2006.

[47] R. Subramanian, Y. Smaragdakis, and G. Loh, "Adaptive caches: effective shaping of cache behavior to workloads," in *Proc. MICRO*, 2006.

[48] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. ISCA*, 2007.

[49] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, J. S. Steely, and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proc. PACT*, 2008.

[50] T. Moseley, "Adaptive thread scheduling for simultaneous multithreading processors." Univ. of Colorado Masters Thesis, 2006.

[51] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen, "Multiple instruction stream processor," in *Proc. ISCA*, 2006.

[52] A. Davis, "Mayfly: a general-purpose, scalable, parallel processing architecture," *Lisp Symb. Comput.*, vol. 5, no. 1-2, 1992, pp. 7–48.

[53] J. K. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?," in *Proc. USENIX*, 1990.

[54] J. C. Mogul, A. Baumann, T. Roscoe, and L. Soares, "Mind the gap: Reconnecting architecture and os research," in *Proc. HotOS*, 2011.

[55] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania, "The Multikernel: A new OS architecture for scalable multicore systems," in *Proc. SOSP*, 2009.

[56] S. Peter, A. Schpbach, D. Menzi, and T. Roscoe, "Early experience with the Barrelfish OS and the Single-Chip Cloud Computer," in *Proc. MARC Symp.*, 2011.

[57] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architecture support for single address space operating systems," in *Proc. ASPLOS*, 1992.

[58] J. S. Chase, M. B.-Harvey, H. M. Levy, and E. D. Lazowska, "Opal: A single address space system for 64-bit architectures," *Oper. Syst. Rev.*, vol. 26, no. 2, 1992.

[59] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proc. HotOS*, 1995.

[60] D. Engler, "The exokernel operating system architecture." Massachusetts Inst. of Tech. PhD thesis, 1999.

[61] J. Mogul, D. Western, J. C. Mogul, and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans. Comput. Syst.*, vol. 15, 1997.

[62] S. Radhakrishnan, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "Nicpic: Scalable and accurate end-host rate limiting," in *Proc. HotCC*, 2013.

[63] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "Senic: A scalable nic for end-host rate limiting," in *Proc. NSDI*, 2014.

[64] J. C. Mogul, J. Mudigonda, J. R. Santos, and Y. Turner, "The nic is the hypervisor: Bare-metal guests in iaas clouds," in *Proc. HotOS*, 2013.

[65] K. Ram, J. Mudigonda, A. Cox, S. Rixner, P. Ranganathan, and J. Santos, "snich: Efficient last hop networking in the data center," *Proc. SANCS*, 2010.

[66] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable persistent memory," in *Proc. SOSP*, 2009.

[67] A. Salz and M. Horowitz, "IRSIM: an incremental MOS switch-level simulator," in *Proc. DAC*, 1989.

[68] 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," *Comp. Arch. News*, 2005.

[69] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. A. Patterson, and K. Asanovic, "Ramp gold: an fpga-based architecture simulator for multiprocessors," in *Proc. DAC*, 2010.

[70] T. Li and L. John, "OS-aware tuning: improving instruction cache energy efficiency on system workloads," in *Proc. IPCCC*, 2006.

[71] B. Wun and P. Crowley, "Network I/O acceleration in heterogeneous multicore processors," *Proc. HotI*, 2006.

[72] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Proc. MICRO*, 2003.

[73] D. Nellans, R. Balasubramonian, and E. Brunvand, "OS execution on multicores: Is out-sourcing worthwhile?," *Oper. Syst. Rev.*, April 2009.

[74] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Proc. IISWC*, 2008.

[75] J. L. Henning, "SPEC CPU2006 benchmark descriptions," in *Proc. Comp. Arch. News*, 2005.

[76] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A benchmark suite of bioinformatics applications," in *Proc. ISPASS*, 2005.

[77] U.S. Environmental Protection Agency - Energy Star Program, *Report to congress on server and data center energy efficiency - Public Law 109-431*. [Online], Available: `http://www.energystar.gov`.

[78] Sun Microsystems Inc., *The SPARC Architecture Manual Version 9*. [Online], Available: `http://www.sparc.org/standards/SPARCV9.pdf`.

[79] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. MICRO*, 2007.

[80] R. Strong, J. Mudigonda, J. Mogul, N. Binkert, and D. Tullsen, "Fast switching of threads between cores," *Oper. Syst. Rev.*, April 2009.

[81] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Dynamically managing the communication-parallelism trade-off in future clustered processors," in *Proc. ISCA*, 2003.

[82] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," in *IEEE Micro*, vol. 23, 2003, pp. 84–93.

[83] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *Proc. ISCA*, 1999.

[84] D. Chiou, "Extending the reach of microprocessors: column and curious caching.." Massachusetts Inst. of Tech. PhD Thesis, 1999.

[85] M. Powell, A. Agrawal, T. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via selective direct-mapping and way prediction," in *Proc. MICRO*, 2001.

[86] Author Unknown, *Completely Fair Queueing (CFQ) Scheduler*. [Online], Available: `https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt`.

[87] J. Corbet, *Deadline Scheduler*. [Online], Available: `http://www.linuxfoundation.org/news-media/blogs/browse/2014/01/deadline-scheduling-314`.

[88] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proc. FAST*, 2012.