A Case for Increased Operating System Support in Chip Multi-Processors

David Nellans, Rajeev Balasubramonian, Erik Brunvand
School of Computing, University of Utah
Salt Lake City, Utah 84112
{*dnellans, rajeev, elb*}*@cs.utah.edu*

Abstract

We identify the operating system as one area where a novel architecture could significantly improve on current chip multi-processor designs, allowing increased performance and improved power efficiency. We first show that the operating system contributes a non-trivial overhead to even the most computationally intense workloads and that this OS contribution grows to a significant fraction of total instructions when executing interactive applications. We then show that architectural improvements have had little to no effect on the performance of the operating system over the last 15 years. Based on these observations we propose the need for increased operating system support in chip multiprocessors. Specifically we consider the potential of a separate Operating System Processor (OSP) operating concurrently with General Purpose Processors (GPP) in a Chip Multi-Processor (CMP) organization.

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 workstations have progressed from the venerable single core 486/33, released in 1989, to the current IBM Power 5 and Intel dual core Pentium 4 EE. Process fabrication size has shrunk from 1 micron (486/33) down to 90 nanometers (Pentium Prescott) and is expected to continue down to 65 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]. Transistor count and resultant die size has exploded from 1.2 million transistors on the 486/33 to more than 275 million in the Power 5. 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. What is striking about these performance improvements, however, is that while general application performance has improved approximately 200x in this time period, 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.

One contributor to this effect is that the operating system is 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 architectural model, such as Simplescalar [2], 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 [3, 4, 5, 6] but often require modification to the operating system or do not provide cycle accurate numbers. While several studies suggest that the operating system has a significant role in commercial workload performance [7, 8, 9, 10], the effect of the operating system on varying workloads is still largely unexplored. Our approach in this paper is twofold: we simulate a variety of applications with a full system simulator to measure the portion of system cycles spent in OS tasks, and we take actual measurements on real machines to try to understand how performance has scaled across different implementations of the same instruction set architecture.

While Redstone et al. [7], have shown the OS can account for over 70% of the cycles in commercial workloads, we contribute to this body of work by examining a variety of workloads. We examine computationally intensive workloads to determine the minimum overhead the operating system contributes to workloads. This provides a baseline estimate of the potential inaccuracy of architectural mod-

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%

Table 1: Operating System Contribution In Various Workloads

els such as Simplescalar compared to Simics coupled with University of Wisconsin's Multifacet Gems Project [11]. We show that minimally, the operating system will contribute 5% overhead to almost all workloads. Workloads which utilize system calls or trigger device interrupts have increased OS overhead, with the suite of interactive applications in our study requiring at least 38% OS overhead. I/O intensive applications often even have operating system overheads which overshadow their userspace components.

With operating system overhead being a significant proportion of instructions executed in many workloads, the performance of the operating system is a critical factor in determining total system throughput. It is currently hard to measure operating system performance due to the lack of cycle accurate simulator support. Instead we measure the effects of architectural improvements by measuring performance improvement on real machines for key components such as context switch time, and system call latency. We find that compared to user codes (which are typically the codes targeted by processor optimizations), operating system performance improvement is consistently lower by a factor of 4 or 5.

We also briefly examine the issue of operating system interference when powering down processors on modern machines. We show that offloading interrupt handling to specialized hardware has the potential to allow more aggressive power saving techniques for longer durations than currently possible.

Having shown that operating systems significantly underutilize modern processors and can contribute a large portion of executed instructions to workloads, we explore the possibility of executing the operating system on a semi-custom processor in a heterogeneous chip multi-processor organization. First we identify three key criteria that make the operating system a good candidate for hardware offload. We then discuss the performance implications of such an organization, followed by situations in which significant power savings could also be achieved.

2 OS Contribution in Workloads

Previous work [7, 9, 10] has shown that the operating system can contribute a significant amount of overhead to commercial workloads. To our knowledge no work has been done to survey the amount of operating system overhead that is present in a larger variety of workloads. Determining the minimal amount of operating system overhead present in all workloads is useful in estimating the amount of possible error in simulations that do not model these effects. In workloads that have significant operating system overhead, such as databases and web server workloads, performance can depend as much on the operating system as on user code. For these applications, knowing operating system overhead is critical for performance tuning. Finally, very rarely are interactive applications examined when architecting performance improving features because they typically under-utilize modern processors. Because of their prevalence, it helps to explore this application space to determine if architectural improvements can reduce power consumption while maintaining current performance levels.

2.1 Methodology

Most architectural simulators do not execute operating system code because instrumenting external interrupts and device I/O models is a non- trivial task in addition to significantly slowing down the overall simulation speed. Simics, a full system simulator from Virtutech [4], provides full system simulation by implementing such device models allowing it to execute an unmodified Linux kernel. To maintain simulation speed Simics implements functional models of these devices as well as the processor. The lack of an accurate timing model limits the performance evaluation possible using this simulator. Functional modeling is advantageous however because it provides execution fast enough to model not only the operating system but interactive applications using X-Windows.

Simics' functional model of a Pentium processor allows us to examine the processor state bit to determine if the processor model was in privileged mode (supervisor mode) or user mode when executing each instruction. This privilege separation allows us to track which instructions were executed within the operating system. This tracking method also allows us to track system calls which execute within the user process' address space but are considered operating system functionality.

For our simulated system we used the default Redhat 7.3 disk image provided by Virtutech and left all the default system daemons running. Running with background daemons disabled would not portray a typical system in most cases which often require remote filesystem services, re-

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%

Table 2: SPECcpu2000 Instructions Executed in Supervisor Mode

mote procedure calls invocation, as well as remote login. We recognize that computing installations often heavily optimize the set of system daemons running on their machines, but we do not attempt to model the vastly different subsets of services utilized in enterprise deployments. While system daemons typically do not contribute heavily to overall instruction counts, they do cause sporadic interrupts which can cause scheduling changes and I/O activity.

Typically architectural level simulations are only run for several million cycles due to the slowdown of simulation. We found significant variation when looking at sample sizes of only 10 million instructions due to the OS performing periodic maintenance operations such as clearing write buffers, even when the workload had reached a steady state. To remedy this variation we allowed all benchmarks to run for 10 billion instructions prior to taking measurements. All numbers cited are the average of 10 sequential runs of 100 million instructions which reduced statistical variation to negligible levels.

2.2 CPU Intensive Workloads

Computationally intense workloads are used to evaluate micro-architectural improvements because they maximize computation and minimize the effects of memory and I/O performance as much possible. These benchmarks often represent scientific workloads which 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 [12] 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 1 shows the percentage of instructions executed in supervisor mode by these benchmarks. Table 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%. The ByteMark benchmark skews these results strongly however and we believe the average of 5.19% for the SPEC benchmarks is a more realistic minimal overhead.

2.3 I/O Intensive Workloads

Computationally intense benchmarks are a good way to test architectural improvements 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 [13] and netperf, a TCP performance benchmark, to measure OS contribution in what we expected to be OS dominated workloads. Table 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 surprisingly high operating system contribution of 97.16%. These benchmarks confirm that I/O processing is almost entirely OS dominated and support the work of Redstone et al. [7] who have shown that for workloads such as Apache the operating system can contribute more than 70% of the total cycles.

2.4 Interactive Workload

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 slow-

ing 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 reproducible across multiple runs due to spurious interrupt timing, we believe smoothing the operating system effects across billions of instructions accurately portrays the operating system contribution. This interactive workload typically only utilized about 32.91% of the processor's cycles and spent 38.04% of these instructions within the OS.

3 Decomposing Performance Improvements

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) [14, 15]. 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.

To determine the architectural improvement, A, when moving from the 486 to the Pentium 4 we must accurately determine the total performance improvement P and then factor out the technological improvement T. To achieve accurate values for P we installed identical versions of the Linux kernel and system libraries, kernel 2.6.9 and gcc 3.4.4, on both the machines. Each machine was individually tuned with compiler optimizations for best performance. Using the same kernel and system libraries helps minimize any possible performance variation due to operating system changes even from minor revisions.

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

Table 3: Speedup from 486 33MHz to Pentium 4 3.0GHz

Attempting to discern the architectural improvements of only the microprocessor required that we use benchmarks that minimize the use of external hardware in a performance critical manner. We also chose benchmarks with working sets that fit in the limited main memory of our 486 test machine. This ensured that the machines would not regularly access the swap partition which could provide misleading performance numbers. We chose the Linux kernel compile because there is enough parallelism present by creating multiple jobs that compilation can fully utilize the processor while other jobs are waiting on data from the I/O system. Care has to be taken to create only the minimum number of jobs necessary to fully utilize the processor or we can introduce significantly more context switching than necessary. N + 1 jobs is typically the ideal number of jobs, where N is the number of processors available when compiling source code.

Table 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 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.

3.1 Operating System Performance

In section 2 we showed that the operating system contributes non-trivial overhead to most workloads and that

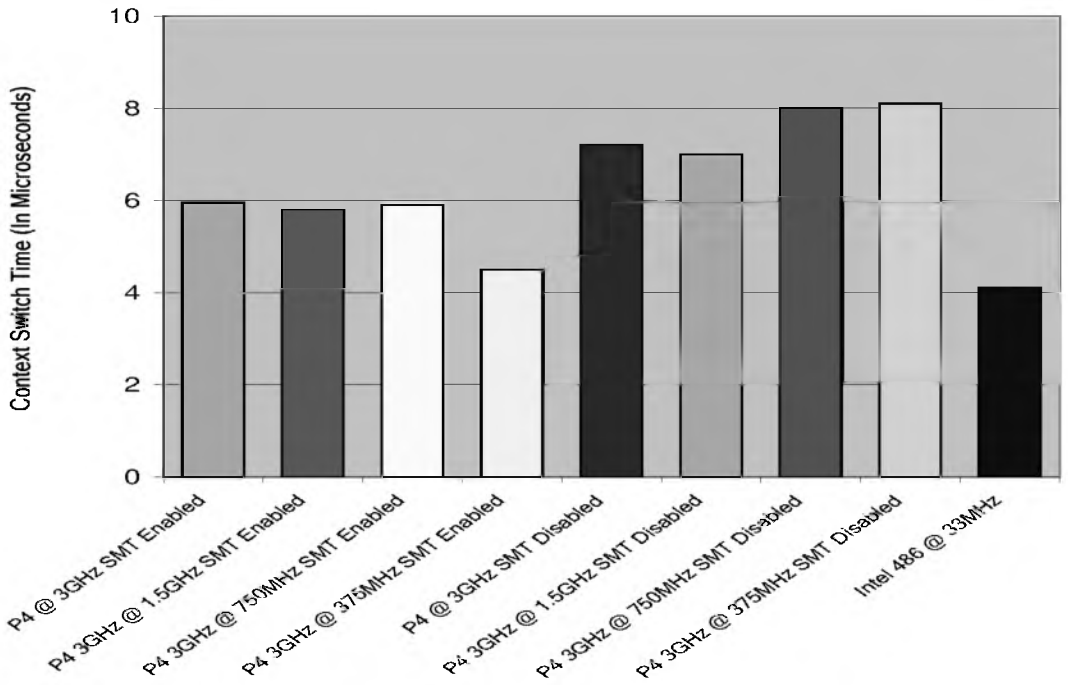

Figure 1: Context Switching Speed Normalized for Clock Frequency

some workloads, such as those which are interactive or I/O intensive, can have significant portions of their total instruction counts occur within the operating system. For these applications operating system performance is critical to achieve good workload performance. Table 3 shows that while total application performance has increased 192.2 times in the last 15 years, operating system performance has only increased 46.04 times. Stated another way, application codes that can take full advantage of modern architectural improvements achieve four times more performance than operating system codes that can not.

Table 3 shows only a limited set of benchmarks because our 486 was restricted to this set due to its limited memory. Additionally the UNIX time command does not provide the timing fidelity necessary to have good confidence in the low number of seconds reported for the system time on the Pentium 4. To validate the OS speedup results in Table 3 we perform two other experiments in subsequent sections which each independently support the conclusion that architectural improvements have helped application codes significantly, but are from 4 to 10 times less effective on operating system codes.

3.2 Context Switch Performance

As single threaded applications are redesigned with multiple threads to take advantage of SMT and CMP based processors, the context switch, an already costly operation, will become even more critical. Operating system designers have focused on reducing the cost of a context switch for years, while much less attention has been paid in the architecture community [16, 17]. To determine the improvement in context switch time over the last 15 years we measure context switch cost using a Posix locking scheme similar to the one described in [18]. In this method two threads are contending for a single Posix lock, similar in style to traditional methods of passing a token between threads. The Posix lock is used instead of a traditional UNIX pipe because Posix locks are one of the lowest overhead synchronization primitives currently available in Linux.

Figure 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 clock frequency for a given architecture. Our Pentium 4 running at 3.0GHz supports fre-

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	.63
fcntl64	62	1	62	5.63
fork	15,985	8,187	2	.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	3
newuname	88	2	44	4
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	.45
time	72	5,585	0	0
waitpid	99,917	3	33305	3027.72
write	2,164	3,418	0	0

Table 4: System Call Speedup from 486 33MHz to Pentium 4 3.0GHz - Time in microseconds

quency 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 operation. Thus we can draw the conclusion that the number of cycles required to context switch in a given microarchitecture is independent of clock frequency.

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 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 saw in Table 3.

3.3 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) [19] 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 3 we also eliminate variation due to particular workload patterns. The Linux Trace Toolkit provides microsecond timing fidelity, thus system calls which take less than 1 microsecond can be reported as taking either 0 microseconds or 1 microsecond depending 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.

Table 4 shows the absolute time for common system calls, absolute speedup, and the speedup due to architectural improvement only, using our technology scaling factor T set at 11. When examining these figures we must be careful to examine the function of the system call before interpreting the results. System calls which 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.

3.4 Interrupt Handling

Interrupt handling is a regular repetitive task that all operating systems perform on behalf of the user threads for which incoming or outgoing data is destined. Using the Linux Trace toolkit we found that the number of interrupts handled by the operating system during a 10 second period on the Pentium 4 was within 3% of the number handled on the 486 during the same 10 second period. These timings were performed with both interactive and computationally intense workloads. The average of these two workloads is cited in Figure 2 which shows these ex-

ternal interrupts are regular in their arrival. In many cases these interrupts are not maskable meaning they must be dealt with immediately and often cause a context switch on their arrival.

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 18000 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 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 [20].

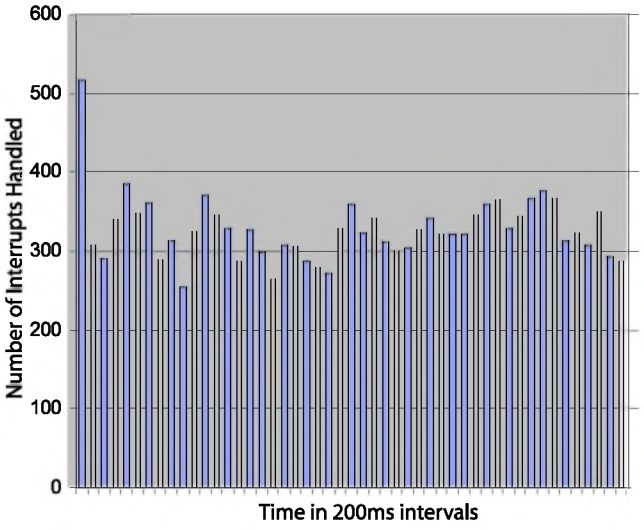

Figure 2: Distribution of Interrupt Handling Over 10 Seconds

4 Proposed Architectural Support for Operating Systems

Hardware specialization via offloading of applications from the general purpose processor has occurred many times for applications such as graphics processing, network processing, and I/O control. These specialized processors have been successful whereas many others, such as TCP off-load, have not. We identify three crucial requirements that an offloaded application must meet to justify the use of specialized hardware. Failure to meet all three of the criteria likely indicates that the potential performance and power benefits gained from hardware specialization will not exceed the communication overhead that is also introduced. Based on these criteria we believe the operating system is a prime candidate for hardware support in a chip multiprocessor configuration.

4.1 Criteria for Hardware Specialization

Constant Execution Applications that contribute a significant portion of cycles to the total workload tend to benefit from hardware specialization. Amdahl's law tells us that we should spend our resources speeding up applications that contribute the largest amount to total system performance. Graphics processors, network handling, and I/O controllers are all examples of applications that are constantly running and consume a non-trivial fraction of a time-shared processor's cycles. In Section 2, we established that the operating system minimally contributes five percent overhead to any workload, and that many applications require at least 38% OS overhead to achieve graphical interactivity. I/O intensive applications can have more than half of their total instructions occur within the operating system.

Inefficient Execution on General Purpose Processors Offloaded applications must stand to gain a decisive performance advantage by being offloaded to a more specialized processor. If no performance gain can be achieved when offloading an application, the additional hardware will typically consume more power to achieve the same total system performance. Conversely, it may also be worthwhile to off-load applications that can be executed with acceptable performance while utilizing significantly less power. In Section 3, we established that operating system performance improvement is lagging behind user code performance improvements on modern machines by four-fold. In the last 15 years, technology improvements have increased the absolute performance of the operating system three times as much as architectural innovations. Having established the inefficiency of operating system execution on modern processors, we believe it will be possible to provide hardware support that will simultaneously save power and increase performance through architectural improvements.

Benefit To Remaining Codes Offloading a specific application to specialized hardware must provide a signifi-

cant benefit to the codes that will remain on the general purpose processor. We have shown that the operating system executes frequently and regularly, with at most 6.2ms between invocations when handling only interrupts. Every invocation of the operating system introduces cache and branch history entries into performance critical structures. Upon return to the user codes, these structures are often no longer at their steady state and require a warm up period before they can be fully utilized again. The interference caused by the OS in these structures has been shown to be worse than the interference caused by other user-level threads [20]. As a result, we believe that eliminating intermittent operating system execution from the main processing unit will result in improved performance for the user portions of application execution even with user applications now causing more direct interference between each other.

4.2 Proposed Architecture

We believe the operating system meets the key criteria that help identify exceptionally good candidates for additional hardware support. With the advent of chip multiprocessors, adding more cores on a single die is becoming commonplace. Future cores will likely have significantly more than two cores, the standard today, on a single die. We propose a heterogeneous CMP such that one core will be customized to handle operating system execution while the remaining cores are left to execute general purpose codes. Other work [21, 22, 23], has explored the possibility of heterogeneous chip multi-processors but to our knowledge our work is the first to propose a heterogeneous CMP that targets a ubiquitous application (the OS) with known poor performance.

Our results in Sections 2 and 3 have shown that, scaled for technology, a classic 5 stage pipeline architecture such as that found on a 486 is surprisingly close in performance to a modern Pentium 4 when executing operating system codes. More startling is that the 486 architecture uses only 1.2 million transistors while the Pentium 4 uses well over 200 million. We believe an architecture not significantly more complex than the 486 could be customized to execute operating system codes as fast or faster than modern general purpose processors at a cost of less than a few million transistors, an insignificant fraction of today's transistor budgets. At similar performance we also believe this semi-custom will do significantly better on metrics which take energy expenditure into account, such as energy delay product. The purpose of this work was to evaluate the need and potential benefits of an architecture of this nature; detailed evaluations of an architecture based on our current observations will be part of future work.

4.3 Potential Performance Benefits

We believe a simple architecture will be able to execute the operating system more efficiently and there are several key architectural features that will improve overall system throughput greatly. The operating system typically executes code segments that are significantly smaller than user codes. In Section 3, we showed that system calls and interrupt handling typically require tens of microseconds to execute upon invocation. These short bursts of instructions do not allow for caches and branch predictors to warm up properly, reducing their effectiveness. Similarly, deep pipelines excel at throughput oriented tasks at the expense of latency. They are also costly when branch mispredictions occur, events that currently happen frequently in operating system codes [20].

Slight modifications to current chip multiprocessor architectures can potentially yield an even more significant performance benefit. Currently, the operating system must examine its run queues to determine which application will be scheduled next. On uni-processors this is a sequential operation with the OS performing this calculation between every two user processes. Offloading the operating system to dedicated hardware would not only allow this function to be precomputed, but could allow the operating system to prefetch context information into a high speed context buffer causing reduced context switch times. Similarly, with a priori knowledge of upcoming contexts, the operating system is in a special position to warm up microarchitectural structures by prefetching instructions and data into the L2 that are likely to be used by the incoming application. We believe that the knowledge the operating system has of contexts can be heavily utilized to reduce these traditional context switching penalties in a variety of ways.

4.4 Potential Power Savings

Dynamic power dissipation has traditionally dominated the energy expended by a processor and techniques such as clock gating and frequency scaling are being utilized to help minimize the impact of this transistor switching. As process sizes shrink, static power begins dominating the total power consumption of microprocessors [24, 25]. Static power dissipation can be mitigated through power gating and voltage scaling, but these techniques require a significant number of cycles to switch between modes. These methods often can not be utilized because the processor is not idle for sufficient intervals of time.

The Pentium 4 processor at 3.8GHz has thermal packaging to support a steady state power dissipation of 115 watts . As die temperature rises, the P4 can modulate its clock on and off to reduce dynamic power consump-

tion by up to 82.5% [1]. Unfortunately, with static power estimated to be nearing half of total power consumption, using only the Pentium 4's on-demand clock modulation, the processor still consumes 68 watts. Eight of these watts are due to dynamic power consumption, while 60 watts are static power loss that cannot be addressed by clock modulation.

In Section 3, we identified interrupt handling as a periodic task with an average interval of 6.8 milliseconds between invocations. On a uni-processor, this period is the longest possible time a processor can perform power gating and reduce static power dissipation. On idle machines, this interval becomes the limiting factor for potential static power savings because the machine must wake up on average every 6.8ms before returning to deep sleep mode. By offloading this interrupt handling to an operating system processor, which we expect to consume a trivial amount of power relative to modern processors, these high wattage cores can be powered down for much longer periods of time resulting in significant total power savings. Note that the power benefits above are in addition to the power benefits of executing OS code on a simple shallow pipeline.

5 Conclusions

We have shown that the operating system minimally poses a 3-5% overhead for all applications and much more for some. Applications which require user interaction utilizing X Windows can incur a much larger operating system overhead of 38%. Applications which have significant I/O requirement can have even larger operating system components. For applications in which the operating system contributes a significant portion of its total instructions the performance of operating system codes is critical for achieving high application performance. The modeling of operating systems in architectural simulation is now critical if we wish accurately predict performance gains in future generations of microprocessors.

In the last 15 years total system performance has improved by almost 200 times, 11 times from technology improvements, based on FO4 delay, and 19.7 times from architectural improvements. The operating system has received the same 11 fold improvement from technology but has seen only a 4 fold improvement from architectural improvements and executes nearly 5 times slower than application codes on modern hardware.

Based on these studies we propose the concept of a heterogeneous chip multiprocessor consisting of a semi-custom operating system processor (OSP) intelligently coupled with one or more traditional general purpose processors (GPP). This OSP is likely to have a significantly more shallow pipeline than current processors which will maximize performance while also increasing energy efficiency. Providing external interfaces to performance critical structures in the GPP such as branch history tables and caches would allow the OSP to actively pre-warm these structures for incoming contexts of which the OSP has a priori knowledge. Recurring tasks, such as interrupt handling, which significantly under-utilize the GPP can now be offloaded to a power efficient OSP and allow more aggressive power down of high wattage cores. We believe these benefits will come at an increased transistor budget of at most a few percent.

References

[1] I. Corporation, "Intel pentium 4 processors 570/571, 560/561, 550/551, 540/541, 530/531 and 520/521 supporting hyper-threading technology." pp. 76–81.

[2] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

[3] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the simos machine simulator to study complex computer systems," *Modeling and Computer Simulation*, vol. 7, no. 1, pp. 78–103, 1997, citeseer.nj.nec.com/rosenblum97using.html.

[4] Virtutech, "Simics full system simulator," January 2004, http://www.virtutech.com.

[5] L. Schaelicke, "L-rsim: A simulation environment for i/o intensive workloads," in *Proceedings of the 3rd Annual IEEE Workshop on Workload Characterization 2000*, Los Alamitos, CA (USA), 2000, pp. 83–89. [Online]. Available: citeseer.ist.psu.edu/schaelicke01architectural.html

[6] M. Vachharajani, N. Vachharajani, D. A. Penry, J. Blome, and D. I. August, "The liberty simulation environment, version 1.0," *Performance Evaluation Review: Special Issue on Tools for Architecture Research*, vol. 31, no. 4, March 2004.

[7] J. Redstone, S. J. Eggers, and H. M. Levy, "An analysis of operating system behavior on a simultaneous multithreaded architecture," in *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 245–256. [Online]. Available: citeseer.nj.nec.com/redstone00analysis.html

[8] J. B. Chen and B. N. Bershad, "The impact of operating system structure on memory system performance," in *Symposium on Operating Systems Principles*, 1993, pp. 120–133. [Online]. Available: citeseer.nj.nec.com/chen93impact.html

[9] 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 *Proceedings of Computer Architecture Evaluation using Commercial Workloads (CAECW)*, February 2002.

[10] L. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese, "Impact of Chip-Level Integration on Performance of OLTP Workloads," in *Proceedings of HPCA-6*, January 2000.

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

[12] B. Magazine, "BYTE Magazine's BYTEmark Benchmark Program http://www.byte.com/bmark/bdoc.htm," 1998.

[13] R. Coker, "Bonnie++ filesystem benchmark http://freshmeat.net/projects/bonnie/, http://www.coker.com.au/bonnie++/," 2003.

[14] 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 *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 14–24.

[15] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *ISCA*, 1997, pp. 206–218. [Online]. Available: citeseer.ist.psu.edu/palacharla97complexity effective.html

[16] T. Ungerer, B. Rob, and J. Silc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, 2003.

[17] M. Evers, P.-Y. Chang, and Y. N. Patt, "Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996, pp. 3–11.

[18] D. E. G. Bradford, "Ibm developerworks, runtime: Context switching, part 1, http://www-106.ibm.com/developerworks/linux/library/l-rt9/?t=gr,lnxw03=rtcsh1," 2004.

[19] K. Yaghmour and M. Dagenais, "Measuring and characterizing system behavior using kernel-level event logging." in *USENIX Annual Technical Conference, General Track*, 2000, pp. 13–26.

[20] T. Li, L. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio, "Understanding and improving operating system effects in control flow prediction," *Department of Electrical and Computer Engineering, University of Texas at Austin Technical Report*, June 2002. [Online]. Available: citeseer.ist.psu.edu/li02understanding.html

[21] R. Kumar, K. I.Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *MICRO, Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[22] S. Balakrishnan, R. Rajwar, M. Upton, and K. K. Lai, "The impact of performance asymmetry in emerging multicore architectures." in *ISCA*, 2005, pp. 506–517.

[23] H. P. Hofstee, "Power efficient processor architecture and the cell processor." in *HPCA*, 2005, pp. 258–262.

[24] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches," in *HPCA*, 2001, pp. 147–158. [Online]. Available: citeseer.ist.psu.edu/yang01integrated.html

[25] J. A. Butts and G. S. Sohi, "A static power model for architects," in *International Symposium on Microarchitecture*, 2000, pp. 191–201. [Online]. Available: citeseer.ist.psu.edu/butts00static.html