Operating Systems and Asymmetric Single-ISA CMPs: The Potential for Saving Energy

Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Partha Ranganathan, Vanish Talwar
HP Laboratories Palo Alto

HPL-2007-140
August 22, 2007*

CMP, multi-core, energy savings, operating systems

CPUs consume too much power. Modern complex cores sometimes waste power on functions that are not useful for the code they run. In particular, operating system kernels do not benefit from many power-consuming features that were intended to improve application performance. We propose using asymmetric single-ISA CMPs (ASISA-CMPs), multicore CPUs where all cores execute the same instruction set architecture but have different performance and power characteristics, to avoid wasting power on operating systems code. We describe various design choices for both hardware and software, describe Linux kernel modifications to support ASISA-CMP, and offer some quantified estimates that support our proposal.

* Internal Accession Date Only

Operating Systems and Asymmetric Single-ISA CMPs: The Potential for Saving Energy

Jeffrey C. Mogul Jayaram Mudigonda Nathan Binkert Partha Ranganathan Vanish Talwar
Jeff.Mogul@hp.com, Jayaram.Mudigonda@hp.com, binkert@hp.com, Partha.Ranganathan@hp.com, Vanish.Talwar@hp.com
HP Labs, Palo Alto, CA 94304

Abstract

CPUs consume too much power. Modern complex cores sometimes waste power on functions that are not useful for the code they run. In particular, operating system kernels do not benefit from many power-consuming features that were intended to improve application performance. We propose using asymmetric single-ISA CMPs (ASISA-CMPs), multicore CPUs where all cores execute the same instruction set architecture but have different performance and power characteristics, to avoid wasting power on operating systems code. We describe various design choices for both hardware and software, describe Linux kernel modifications to support ASISA-CMP, and offer some quantified estimates that support our proposal.

1 Introduction

While Moore's Law has delivered exponential increases in computation over the past few decades, two well-known trends create problems for computer systems: CPUs consume more and more power, and operating systems do not speed up as rapidly as most application code does. Many people have addressed these problems separately; we propose to address them together.

Until recently, designers of high-end CPU chips tended to improve single-stream performance as much as possible, by exploiting instruction-level parallelism and decreasing cycle times. Both of these techniques are now hard to sustain, so recent CPU designs exploit shrinking VLSI feature sizes by using multiple cores, rather than faster clocks. Examples of these *Chip Multi-Processors* (CMPs) include the Sun Niagra processor with eight cores, the quad-core Intel Xeon, and dual-core systems from several vendors.

All commercially-available general-purpose CMPs, as of mid-2007, are *symmetrical*: each CPU is identical, and typically, all run at the same clock rate. However, in 2003 Kumar *et al.* [13] proposed heterogeneous (or asymmetrical) multi-core processors, as a way of reducing power requirements. Their proposal retains the *single-Instruction-Set-Architecture* (single-ISA) model of symmetrical CMPs: all cores can execute the same machine code. They observed that different implementations of the same ISA had order-of-magnitude differences in power consumption (assuming a single VLSI process). They further observed that in a multi-application workload, or even in phases of a single application, one does not always need the full power and functionality of the most complex CPU core; if a CMP could switch a process between cores with different capabilities, one could maintain throughput while decreasing power consumption.

Since the original study by Kumar *et al.*, several other studies [3, 8, 14] have highlighted the benefits from heterogeneity. (Keynote speakers from some major processor vendors have also suggested that heterogeneity might be commercially interesting [2, 22].) However, all these studies looked only at user-mode execution. But we know that many workloads spend much or most of their cycles in the operating system [23]. We also know that operating system (OS) code differs from application code: it avoids the floating point processor, it branches more often, and it has lower cache locality (all reasons why OS speedups lag application speedups on modern CPUs). An *asymmetric single-ISA CMP* (ASISA-CMP) might therefore save power, without reducing throughput, by executing OS code on a simple, low-power core, while using more complex, high-power cores for application code.

The main contribution of this paper is to propose and evaluate the ASISA-CMP model, in which (1) a multi-core, single-ISA CPU includes some "OS-friendly" cores, optimized to execute OS code without wasting energy, and (2) the OS statically and/or dynamically decides which code to execute on which cores, so as to optimize throughput per joule.

To optimally exploit ASISA-CMP, we expect that the OS and the hardware both must change. This paper explores the various design considerations for co-evolving the OS and hardware, and presents experimental results.

2 Related work

Ousterhout [20] may have been the first to point out that "operating system performance does not seem to be improving at the same rate as the base speed of the underlying hardware." He speculated that causes include memory latencies and context-switching overheads.

Nellans *et al.* [19] measured the fraction of cycles spent in the OS for a variety of applications, and re-examined how OS performance scales with CPU performance, suggesting that interrupt-handling code interferes with caches and branch-prediction history tables. They found that many applications execute a large fraction of cycles in the OS, and observed that "a classic 5 stage pipeline [such as] a 486 is surprisingly close in performance to a modern Pentium 4 when executing [OS code]." However, instead of proposing an ASISA-CMP, they suggest adding a dedicated OS-specific core. (It is not entirely clear how far their proposal is from a single-ISA CMP.) They did not evaluate this proposal in detail.

Chakraborty *et al.* [11] proposed refactoring software so that similar "fragments" of code are executed on the same core of a CMP. Their initial study treated the OS and the user-mode application as two coarse-grained fragments, and found speedup in some cases. However, they did not examine asymmetric CMPs or the question of power reduction.

Sanjay Kumar *et al.* [15] propose a "sidecore" architecture to support hypervisor operations. In their approach, a hypervisor is restructured into multiple components, with some running on specialized cores. Their goal was to avoid the expensive internal state changes triggered via traps (e.g., VMexit in Intel's VT architecture) to perform privileged hypervisor functions. Instead, the sidecore approach transfers the operation to a remote core "that is already in the appropriate state." This also avoids polluting the guest-core caches with code and data from hypervisor operations. (The sidecore approach is not specifically targeted at saving energy.)

3 Design overview

Our goal is to address two major challenges for multi-core systems: how to minimize power consumption while maintaining good performance, and how to exploit the parallelism offered by a multi-core CPU. These two issues are closely linked, but we will try to untangle them somewhat.

3.1 Proportionality in power consumption

We want to maximize the energy efficiency of our computer systems, which could be expressed as the useful computational work (throughput) per joule expended. Fan *et al.* [6] have observed that the ideal system would consume power directly proportional to the rate of useful computational work it completes. We refer to this as the "proportionality rule." Such a system would need no additional power management algorithms, except as might be needed to avoided exceeding peak power or cooling limits.

Fan *et al.* argue that "system designers should consider power efficiency not simply at peak performance levels but across the activity range." We believe that the ASISA-CMP approach, with a careful integration of OS and hardware design, can help address this goal. Of course, it is probably impossible to design a system that truly complies with the proportionality rule, especially since many components consume considerable power even when the CPU is idle.

There are at least two ways that one might design a system to address the proportionality rule. First, one could design individual components whose power consumption varies with throughput, such as a CPU that supports voltage and frequency scaling. Second, one could design a system with a mix of both high-power/high-performance and low-power/low-performance components, with a mechanism for dynamically varying which components are used (and powered up) based on system load.

The original ASISA-CMP model, as proposed by Kumar *et al.* [13], follows the second approach, without precluding the first one. In times of light load, activity shifts to low-power cores; in times of heavy load, low-power cores can offer additional parallelism without significant increases in area or power consumption.

In this paper, we extend the ASISA-CMP model by asserting that the ideal low-power core is one that is specialized to execute operating system code. (More broadly, we consider "OS-like code," which we will define in Sec. 4.3.1.) This stems from several observations:

- OS code does not proportionately benefit from the potential speedup of complex, high-frequency cores. Thus, running OS code on a simpler core is a better use of power and chip area.
- Most computer systems (with certain exceptions, such as scientific computing) are often idle. If we could power down complex CPU cores during periods when they would otherwise be idle, we could improve proportionality.

The designs explored in this paper include:

- Multi-core CPUs with a mix of high-power, high-complexity application cores, and low-power, low-complexity OS-friendly cores.
- Operating system modifications to dynamically shift load to the most appropriate core, and (potentially) to power down idle cores.
- Modest hardware changes to improve the efficiency of core-switching.

Of course, the CPU is not the only power-consuming component in a system, and ASISA-CMP does not address the power consumed by memory, busses, and I/O devices, or the power wasted in power supplies and cooling systems. Therefore, even if the CPU were perfectly proportional, the entire system would still fail to meet the proportionality rule. However, as long as CPUs represent the largest single power draws in a system (see Sec. 3.3.1), improving their proportionality is worthwhile.

3.2 Core heterogeneity

The promise of ASISA-CMP depends critically on two facts of CPU core design: (1) for a given process technology, a complex core consumes much more power and die area than a simple core, and (2) a complex core does not improve OS performance nearly as much as it improves application performance.

Table 1 shows the relative power consumption, performance (in terms of instructions per cycle, or IPC), and sizes of various generations of Alpha cores, scaled as if all were implemented in the same process technology. Clearly, the smallest core delivers significantly more performance per watt and per mm^2. In fact, these performance results were based on the SPEC CPU benchmark suite; since operating system performance generally scales worse than application performance [20], we believe the IPC ratios would be even smaller for OS code.

Table 1: Power and relative performance of Alpha cores

Alpha core	**Peak power**	**Average power**	**Normalized vs. EV4**		
			IPC	**area**	**power**
EV4	4.97W	3.73W	1.00	1.00	1.00
EV5	9.83W	6.88W	1.30	1.76	1.84
EV6	17.8W	10.68W	1.87	8.54	2.86
EV8	92.88W	46.44W	2.14	82.2	12.45

All cores scaled to 0.1 μm; IPC based on SPEC CPU benchmarks
Based on data from Kumar *et al.* [13]

3.3 Complicating issues

Various issues complicate the question of whether we can improve throughput/joule by running OS (or OS-like) code on special a OS-friendly core. We cover many details in subsequent sections of this paper; here, we expose some general questions. Many of these can only be resolved by experimentation (possibly through simulation); we describe our experiments later, in Sec. 7.

The two key issues, as mentioned above, are the relative power consumption levels for various system components, and the relative performance costs and benefits of switching cores. In order for ASISA-CMP to pay off, we require that it does not reduce performance faster than it reduces power consumption.

3.3.1 How important is CPU power?

Any real system includes multiple components that draw power, and ASISA-CMP will not significantly change the energy consumption of components other than the CPU. In fact, if ASISA-CMP increases the time required to complete a job (or set of jobs), the resulting increase in energy consumed by other system components may outweight the savings from the CPU cores.

Component power consumption varies tremendously across the variety of computer systems in use. In particular,

Table 2: Example power budgets for two typical systems

System	Watts						
	Tot.	CPU	(%)	Mem	Disk	PCI	Other
Blade servers (all with multiple CPUs; after [16])							
Small	248	70	28%	48	10	50	70
Med.	442	170	39%	112	10	50	100
Large	1025	520	51%	320	10	75	100
Laptop (after [17])							
Idle	13.1	2.0	15%	0.4	0.6	N.A.	10.1
Busy	25.8	13.4	52%	1	1	N.A.	10.3

laptops and servers have vastly different balances between components. Table 2 shows a power breakdown for several typical systems. The blade server results, taken from [16], show "nameplate" (maximum) power budgets, and all have either two or (for the "Large" configuration) four CPUs. The laptop results, taken from [17], show measured results for an idle system and for one running the PCMark CPU benchmark; in both cases, Dynamic Voltage Scaling was disabled and the screen was at full brightness.

In all cases, except for the idle laptop, CPU power consumption was the largest single component of system power consumption. For the Large server and the busy laptop, the CPU (or CPUs) consumed slightly more than half of the total power. This suggests that techniques, such as ASISA-CMP, that address CPU power consumption can have meaningful effects on whole-system power consumption.

3.3.2 How does ASISA-CMP affect performance?

ASISA-CMP can affect performance in several ways:

- **Running OS code on a slower CPU**: By design, ASISA-CMP concedes some performance by running OS code on a slower core. As argued in Sec. 3.2, this slowdown might be minimal. However, an application that spends much of its time in the OS could see a significant performance decline.
- **Core switching costs**: ASISA-CMP inherently moves a thread of execution from one core to another for certain system calls. Core-switching creates longer code paths for these system calls, and adds state-saving overhead.
- **Cache affinity vs. cache interference**: We assume that the cores in a CMP CPU share a single L2 cache but have private L1 caches. Core-switching could affect cache performance in at least two ways: it could harm cache affinity, by requiring cache lines (e.g., for the data buffer of a **write** system call) to move between L1 caches, or it could reduce cache interference, by keeping some OS code and data out of the application core's cache.
- **Available parallelism**: Given that the incremental cost (in power and area) for adding an OS core to a CMP is much lower than for an application core (see Sec. 3.2), if there is available parallelism in the workload that extend to OS processing, an ASISA-CMP CPU could support more parallelism than a symmetric CMP CPU with similar power and area. For example, a parallel "make" command might benefit from having an OS core run I/O processing while the application core is dedicated to an optimizing compiler. Not all workloads will have this kind of parallelism, of course.

3.3.3 Idle time

We have described the example in Figure 1 as if the application's system call does useful work. However, it could also be blocked waiting for some external event, such as disk I/O or the arrival of a Web request. Numerous studies (e.g., [6, 21]) have shown that most computers are idle most of the time. Therefore, as pointed out by Fan *et al.* [6], a useful design for meeting the portionality rule must significantly reduce power consumption during idle periods.

ASISA-CMP offers the option of powering down the high-power application core(s), while maintaining OS functions on a low-power core. For example, the arrival of a new Web (HTTP/TCP) connection normally precedes the arrival of the actual HTTP request by at least one network round-trip time (typically on the order of milliseconds or more). This would allow an OS core to handle the initial TCP connection request and then awaken the application core soon enough to process the HTTP-level request without any delay.

3.4 Competing approaches

Given that the goal of ASISA-CMP is to improve performance (in terms of throughput/joule), and it would require changes to the design of CPU chips, we have to compare it against possible competing approaches.

Other potential alternatives include:

- **Complex-core with dynamic Voltage/Frequency**

scaling: This could be especially effective at saving power in systems with lots of idle time. However, we suspect that the lowest-power mode for V/F scaling on a complex core is still much higher than the power drawn by an ASISA-CMP CPU with the application core turned off.

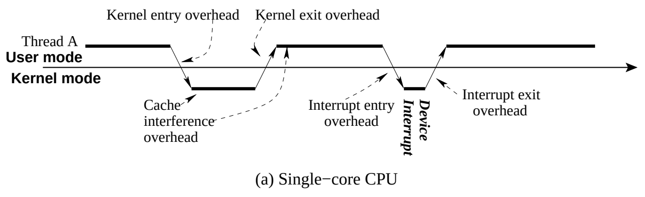
- **Complex-core with power-down on idle**: In such a system, a idle core would wake up on any external interrupt. This approach could also be especially effective at saving power in systems with lots of idle time, especially if the system is not forced to wake up on every clock tick even if there is no work to do. However, this approach only works if the system is truly idle for moderately long periods; a server system handling a minimal rate of request packets, for example, might never stay idle for long enough.
- **Lots of simple cores**: A CPU with lots of simple cores, each of which could be independently power off, might allow a close approximation to the proportionality rule. However, this configuration can only get reasonable throughput (and thus reasonable throughput/joule for the full system) if we can solve the general parallel programming problem – a problem that has been elusive for many years. Amdahl's Law may be an inherent limit to this approach.
- **Some cores with specialized ISAs**: Others have explored this alternative [19]. We have ruled it out for this paper, because we believe that a single-ISA approach makes it much easier to develop and maintain an operating system, and because we have no way to simulate multiple ISAs in one system.

4 Software issues

We believe that the ASISA-CMP approach can be applied to OS code in several ways, including:

1. Dynamically switching between cores in OS kernel code; see Sec. 4.1.
2. Running virtualization “helper processing” on OS-friendly cores; see Sec. 4.2.
3. Running applications, or parts thereof, with “OS-like code” on OS-friendly cores; see Sec. 4.3

To date, we have focussed all of our implementation and experimental work on the first approach.



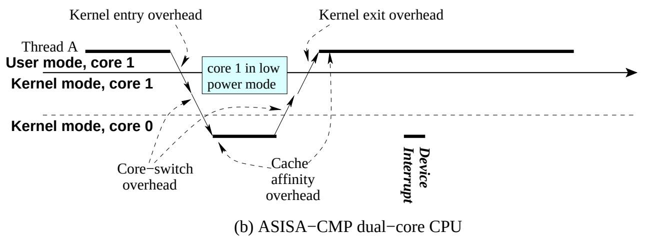


Figure 1: Example without thread-level parallelism.

4.1 Dynamic core-switching for OS kernel code

In any multiprocessor system, performance depends on whether there is enough available parallelism to keep all processing elements busy. Specifically in an ASISA-CMP-based system, there are two ways to optimize throughput/joule:

1. If the system is underutilized: shift OS load to low-power cores, and power down high-power cores, whenever possible.
2. If the system has available parallelism: shift OS load to low-power cores, while keeping the high-power cores as busy as possible with application code.

First, consider the case where there is no available application-level parallelism. Figure 1 illustrates a simple example for an application with just one thread. Figure 1(a) shows a brief part of the application's execution on a single-core CPU. When the application thread makes a system call, execution moves from user mode to kernel mode and back. Figure 1(b) shows the same execution sequence on a two-core ASISA-CMP system. In this case, when the application thread makes the system call, the kernel (1) transfers control from the “application core” (core 1) to the “OS core” (core 0); (2) puts core 1 in low-power mode; (3) executes the system call on core 0; (4) wakes up core 1; and (5) transfers control back to core 1.

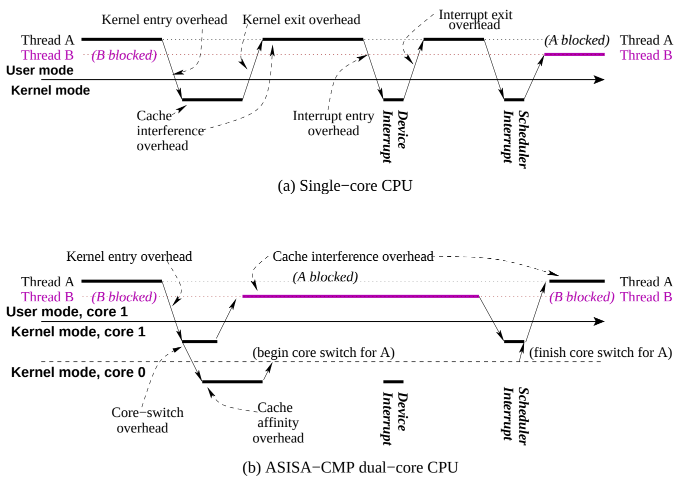


Figure 2: Example with thread-level parallelism.

If the OS core draws significantly less power than the application core while not significantly reducing performance on OS code, and *if* there were no overheads for switching cores and for changing the power state of core 0, then the ASISA-CMP system would have higher throughput per joule than the single-core system. These two "ifs" are two of the key questions for this paper: are there real benefits to running OS code on specialized cores, and are the overheads small enough to avoid overwhelming the benefits?

Figure 1(a) and (b) also show what happens on the arrival of an interrupt. In the single-core system, application execution is delayed both for the actual execution of the interrupt handling code in the OS, and also for interrupt exit and entry. In the ASISA-CMP system, the application continues to run without delay (except perhaps for memory-access interference). This would probably be true for any multicore system, but in the ASISA-CMP approach, interrupt handling happens on a power-optimized core, rather than on a high-power core.

Note that we compare the dual-core ASISA-CMP system against a single-core system, rather than against a symmetric dual-core system, because (in the underutilized case) it seems likely that a dual-core CMP would consume more power than a single-core system, without any increase in throughput.

Next, consider the case where there is application-level parallelism. Figure 2 illustrates another simple example, for an application with two runnable threads, thread A and thread B, where thread A is running at the start of the example, and again the thread makes a system call. Figure 2(a) shows the single-core case: while thread A is executing in the kernel, thread B remains blocked until the scheduler decides that A has run long enough. Figure 2(a) shows the ASISA-CMP case: when thread A makes its system call, the kernel switches that thread to the OS core (core 0), thread B can now run on the application core (core 1).

Again assuming that the switching overheads are not too large, the use of ASISA-CMP increases the utilization of the application core, because OS execution is moved to a more appropriate core.

4.2 Running virtualization helpers on OS-friendly cores

So far, we have discussed the operating system as if it were a single monolithic kernel. Of course, many operating systems designs, both in research and as products, have been decomposed; for example, into a microkernel and a set of server processes. Sec. 4.3 discusses the possibility of running daemon processes on a OS-friendly core.

However, a different kind of decomposition has become popular: the creation of one or more virtualization layers between the traditional operating system and the hardware. These layers basically execute OS kernel code, but in a different protection domain from the "guest" OS. We believe these are clear candidates for execution on OS-friendly cores. They also tend to be bound to specific cores, which eliminates the performance overhead of core-switching.

Note that the use of current virtual machine monitors (VMM) might undermine the use of dynamic core switching for kernel code, because a guest operating system running in a virtual machine might not be able to bind a thread to a specific core. It is possible that a VMM's interface to the guest OS could be augmented to expose the actual cores, or at least to expose the distinction between application cores and OS-friendly cores, but we have not explored this issue in detail.

- **Xen's Domain 0:** In Xen [4], and probably in similar VM systems, one privileged virtual machine is used to multiplex and manage the I/O operations issued by the other virtual machines. This "Domain 0" probably would be most power-efficient if run on an OS-friendly core.

- **I/O and network helper threads:** Several researchers have proposed running I/O-related portions of the virtualization environment on separate processors. McAuley and Neugebauer [18] proposed "encapsulating operating system I/O subsystems in Virtual Channel Processors," which could run on separate processors. Regnier *et al.* [24] and Brecht *et al.* [5] have proposed running network packet processing code on distinct cores. For both of these approaches, OS-friendly cores would be a good match for optimizing power and performance.

4.3 Running OS-like code on OS-friendly cores

Our hardware design is based on the observation that "OS code" behaves differently from user code. However, the definition of an operating system is quite fluid; the same code that executes inside the kernel in a monolithic system, such as Linux, may execute in user mode on a microkernel. Thus, we should not limit our choice of which code to execute on OS-friendly cores based simply on whether it runs in kernel mode.

Typical systems include much code that shares these characteristics with actual kernel code:

- **Libraries:** much of user-mode code actually executes in libraries that are tightly integrated with the kernel. In microkernels and similar approaches, it might be hard to distinguish much of this code from traditional kernel code. So, one might expect some library code to run most efficiently on an OS-friendly core. On the other hand, we suspect that this would require core-switching far too frequently, and it might be hard to detect when to switch.
- **Daemons:** most operating systems (but especially microkernels) use daemon processes to execute code that does not need to be in the kernel, or that needs to block frequently. This code might also be run most efficiently on an OS-friendly core, and could easily be bound to the appropriate core (most operating systems allow binding a process to a CPU).
- **Servers:** Some applications, such as Web servers, might fall into the same category as daemons. However, cryptographic code might run better on normal cores, and so a secure Web server might have to be refactored to run optimally on an ASISA-CMP.

Note that if we use ASISA-CMP to execute entire OS-like application processes (or, at least, threads) on OS-friendly cores, this eliminates the performance overhead of core-switching.

4.3.1 Defining and recognizing "OS-like" code

If it makes sense to run OS-like application code on OS-friendly cores, how does the OS decide which applications to treat this way? Programmers could simply declare their applications to be OS-like, but this is likely to lead to mistakes. Instead, we believe that through monitoring the appropriate hardware performance counters, the OS can detect applications, or perhaps even threads, with OS-like execution behavior.

Automated recognition of OS-like code demands a clear definition of the term. While we do not yet have a specific definition, the main characteristics of OS-like code are likely to include the absence of floating-point operations; frequent pointer-chasing (which can defeat complex data cache designs); idiosyncratic conditional-branch behavior [7] (which can defeat branch predictors designed for application code), and frequent block-data copies (which can limit the effectiveness of large D-caches).

Many of these idiosyncratic characteristics could be visible via CPU performance counters. (Zhang *et al.* [26] have suggested that performance counters should be directly managed by the OS as a first-class resource, for similar reasons.) We are not sure, however, whether support for ASISA-CMP will require novel performance counters.

We also suspect that an adaptive approach, based on tentatively running a suspected OS-like thread on an OS-friendly core, and measuring the resulting change in progress, could be effective in determining whether a thread is sufficiently OS-like to benefit. This approach requires a means for the kernel to determine the progress of a thread, which is an open issue; in some cases, the rate at which it issues system calls could be a useful indicator.

4.3.2 Should asymmetry be exposed to user-mode code?

Kernels already provide processor-affinity controls (e.g., **sched_setaffinity** on Linux). With asymmetrical cores, the kernel might also expose core configuration data, to allow OS-like user code to select the right core.

5 Implementation of dynamic core-switching

For the preliminary experiments in this paper, we made the simplest possible changes to Linux (version 2.6.18) to allow us to shift certain OS operations to an OS-friendly core. We assumed a two-core system, and did not add code to power-down (or slow down) idle cores.

Kernels typically execute two categories of code: "Bottom-half" interrupt-handler code, and "top-half" system-call code. We used different strategies for each.

5.1 Interrupt-handler code

We believe that interrupt code (often known as "bottom half" code) should always prefer an OS-friendly core. The OS can configure interrupt delivery so as to ensure this.

Linux already provides, via the `/proc/irq` directory in the `/proc` process information pseudo-filesystem, a way to set interrupt affinity to a specific core. We do this only for the NIC; most other devices have much lower interrupt rates, and the timer interrupt must go to all CPUs.

It is possible that the interrupt load could exceed the capacity of the OS core(s) in a CPU. If so, should this load be spread out over the other cores? This might improve system throughput under some conditions, but it might be better to confine interrupt processing to a fixed subset of cores, as a way to limit the disruption caused by excessive interrupts. In our experiments, we have statically directed interrupts to the (single) OS core.

5.2 System calls

We faced two main challenges in our implementation of core switching for system calls: how to decide when to switch, and how to reduce the switching costs. Switching cores takes time, because

1. switching involves executing extra instructions;
2. switching requires transferring some state between CPUs;
3. if the target core is powered down, it could take about a thousand cycles to switch [13]. Note that in our current implementation, we do not attempt to power-down idle cores; we defer further discussion of this issue to Sec. 5.6.

Given a significant cost for core-switching, the tradeoff is only beneficial for expensive, frequent system calls (e.g., **select** or perhaps **open**), but not fast or rare calls (e.g., **getpid** or **exit**). (Sec. 7.2 describes simple measurements of relevant costs.) Further, for some system calls the decision to switch should depend on how much work is to be done. A **read** system call with a buffer size of 10 bytes should not switch, whereas one with a buffer size of 10K bytes probably should.

Our basic approach to core-switching is to modify certain system calls so that the basic sequence is:

- do initial validation of arguments
- **Decide whether there is enough work to do to merit switching**
- **If so, invoke a core-switching function to switch to an OS core**
- do the bulk of the system call
- **If we switched codes, core switch back to an application core.** We return, if possible, to the original core, to preserve cache affinity.
- finish up and return

The steps in bold are the modifications we made. The details differ slightly for each call that we modified, but generally involve only a few lines of new code for each call.

In our current implementation, we core-switch on these system calls: **open, stat, read, write, readv, writev, select, poll,** and **sendfile.** For **read**, **write**, and similar calls, we arbitrarily defined "enough work" as 4096 bytes or more; we have not yet tried to optimize this choice.

We have two different implementations of the core-switching function, **asisaSwitchCores**: a very slow design that works, and a faster design that, unfortunately, we were unable to debug before the submission deadline. We describe each version below.

In either case, we dedicate one or more cores as *OS cores*, to execute only OS code, and we modified the kernel to maintain bitmaps designating the OS cores and the application cores.

5.3 Core-switching via the Linux migration mechanism

Linux already includes a load-balancing facility that can migrate a thread from one CPU to another [1]. For our initial implementation of **asisaSwitchCores**, we used this mechanism. This gave us a simple implementation: we block the calling thread, make it eligible to run only on the target destination CPU, place it on a queue of migratable processes, and then invoke Linux's per-CPU migration thread.

This is an expensive procedure, since it involves a thread-switch on the source CPU, and invokes the scheduler on both the source and destination CPUs ... and the entire procedure must be done twice per system call.

5.4 Core-switching via a modified scheduler

In an attempt to speed up core-switching, we wrote a version of **asisaSwitchCores** that directly invokes a modified, somewhat simplified version of the Linux scheduler. Linux allows a thread (running in the kernel) to call the **schedule** procedure to yield the CPU. We wrote a modified version, **asisaYield**, which deactivates the current thread, places it on a special per-CPU queue for the target CPU, does the usual scheduler processing to choose the next thread for the source CPU, and finally sends an inter-processor interrupt to the destination CPU. (Linux already uses this kind of interrupt to awaken the scheduler when doing thread migration.)

When the interrupt arrives at the destination CPU, it invokes another modified version of the scheduler, **asisaGrab**, which dequeues the thread from its special per-CPU queue, bypassing the normal scheduler logic to choose which thread to run.

5.5 Core switching costs

To quantify the cost of core switching, we modified a kernel to switch on the **getpid** system call. This call does almost nothing, so it is a good way to measure the overhead. We wrote a benchmark that bypasses the Linux C library's attempt to cache the results of **getpid**; the benchmark pins itself to the application core, then invokes **getpid** many times in a tight loop. We wrote similar benchmarks to measure the latency of **select** with both an empty file-descriptor set (fdset) and a one-element fdset which is always "ready", and of **readv** (which reads into a vector of N buffers) reading $N = 16, 64, 512$ blocks of 64 bytes from a file in the buffer cache.

The results in Table 3 were measured on our modified Linux 2.6.18 kernel running on a dual-core Xeon model 5160 (3.0GHz, 64KB L1 caches, 4MB L2 cache) system, with and without core-switching enabled. In general, our known-to-be-slow core-switching code adds slightly over 4 microseconds per system call. The **select** call with an empty fdset, and the **readv** call with $N = 16$, both have essentially no overhead because they do not have enough work to core switch (16 blocks of 64 bytes is well below the 4096-byte threshold we set for "enough work" for **readv**).

Table 3: Core-switching overheads

	Core-switching		
System call	**disabled**	**enabled**	**Overhead**
getpid	84	4183	4099
select/empty	120	121	1
select/non-empty	350	4647	4297
readv $N = 16$	6027	6018	-9
readv $N = 64$	22517	26758	4241
readv $N = 512$	176503	181767	5264

Times are in nsec. per call; trial is 10M invocations of system call (except 10K invocations for **readv**); result is from best of 10 trials.

The per-call overhead for **readv** call $N = 512$ increases slightly over that for $N = 64$, possibly because the larger buffer size (32K bytes) represents a significant number of cache-line transfers between cores, or perhaps because copying it from the user buffer to a kernel buffer causes conflicts in the 64KB L1 cache.

5.6 Powering down cores

Kumar *et al.* [13] assumed that "unused cores are completely powered down, rather than left idle," as a way to minimize wasted power. Given their estimated thousand-cycle cost to power-up a core, this might not be the best choice, especially for an OS core that is handling lots of interrupts or system calls, or where interrupt latency is critical.

How should the kernel decide when to power-down cores? We suggest an adaptive approach:

1. If the OS expects to return quickly to an application, it should keep the application core powered up. (The OS might expect to return "slowly" if the application is blocked on disk I/O, for example.)
2. The OS can track the rates of system calls and interrupts. If the combined rate has been low, it could power-down the OS core after returning (although perhaps the arrival of any interrupt implies a likely increase in the rate of system calls?). If the rate exceeds a threshold, the OS core should remain powered up. (Feedback-based optimization could be useful here.)

One possible alternative to completely powering down a core is to drastically reduce the core's voltage and frequency. Sec. 6.2 discusses idle-CPU power states in more detail.

6 Architectural support

The ASISA-CMP approach exposes a number of hardware design choices, which we discuss in this section. Most of these are open issues, since we have not yet done the simulation studies to explore all of the many options.

6.1 Design of ASISA-CMP processors

How should an "OS processor" in an ASISA-CMP differ from the other processors? We strongly favor the single-ISA model, since it greatly simplifies the design of the software, and because it still allows flexible allocation of computation to cores. But given a single ISA, many choices remain:

- **Non-architectural state:** Different implementations of an ISA often expose different non-architectural state to the operating system. ("Non-architectural state" includes CPU state that might have to be saved when a partially-completed instruction traps on a page fault.) The OS needs to be aware of these differences to be able to support the appropriate emulation.
- **Floating-point support:** Most kernels do not use floating point at all, and so one might design the OS core without any floating-point pipeline or registers. If a thread with FP code somehow did end up executing on the OS core, it could trap into FP-emulation code, preserving the single-ISA view at a considerable performance cost.
- **Caches:** CMPs typically have per-core first-level (L1) caches, and a shared L2 cache. Cache designers have many choices (line size, associativity, total cache size, etc.). OS code tends to have different cache behavior than application code [23]; for example, the kernel has much less data-cache locality. On the other hand, a large instruction cache might capture a lot of OS references. An OS core could have smaller caches overall, or it could have a larger L1 I-cache at the expense of a smaller L1 D-cache.
- **Pipeline:** Modern cores have complex pipelines (one version of the Pentium 4 had 31 stages; more recent systems have somewhat fewer stages). Deep pipelines work well for applications with predictable or infrequent branch behavior, but badly for code with frequent and hard-to-predict branches, as is typical of OS code. An OS core can potentially have a simple pipeline and still achieve close to the performance of a much more complex pipeline, but with significant power and area savings.
- **Branch-prediction tables:** Branch prediction (BP) helps CPUs avoid stalling to determine if a branch will be taken or not. BP performance depends on the size of the CPU core's prediction tables. While we know of no studies addressing whether kernel-only execution would be optimized by a different table size than for mixed kernel+user execution, Gloy *et al.* [7] reported that one should not base simulations of BP on user-only traces if the kernel accounts for even 5% of execution time. This result suggests that an OS-friendly core might need a different BP table than a general-purpose core, although it is unclear whether an OS-friendly core could use smaller tables. Their result also suggests that keeping OS and user code on separate cores would improve branch prediction.
- **Number of OS cores:** While this paper mostly assumes an ASISA-CMP with a single OS-friendly core, there is no inherent reason why there should be just one. Future CMPs might have dozens of cores [12], so an ASISA-CMP designer would probably have to choose the appropriate fraction of OS-friendly cores to optimize power vs. performance for a range of anticipated workloads. (One might imagine a small family of CPU products that differ only on this axis, for different markets.)
- **Proximity to I/O:** A system that tries to execute OS code on an OS-friendly core will, in most cases, use that core for I/O operations. CPU designers are moving towards on-chip integration of I/O hardware (e.g., PCI Express controllers, HyperTransport, or Intel's CSI). While these features will probably not soon be integrated into individual cores, placing the OS-friendly core(s) near the on-chip I/O components should reduce wire lengths, improving performance and reducing power consumption.

6.2 Idle-CPU power states

Ideally, we want to be able to put an idle core into a zero-power state with no delay for entry or exit. In reality, the exit transition takes a relatively long time; we may have to settle for transitioning the core into a low-power state. For example, the Advanced Configuration and Power Interface standard [10] specifies multiple "C-states" progressively increasing in aggressiveness of power management. These use a wide variety of options, including gating, voltage and frequency control, architectural throttling (e.g., 4-way issue to 1-way issue), memory sleep states, controller and coherence shut-down, etc.

Open questions include:

- Is there a fundamental tradeoff between the power consumption in a low-power state and the delays for entering and leaving that state?
- What are the fundamental limits to the speed of these transitions? Is there room for further innovation in support of rapid power-level changes, and leakage power reduction in low-power states?
- What new mechanisms might help speed up these transitions?
- Would ASISA-CMP yield better overall performance (throughput/joule) if it used a deep idle state with high transition costs, or a shallow idle state with low transition costs? If the choice depends on the application mix, as seems plausible, could the the OS decide dynamically between these approaches (e.g., by estimating duration of the next idle period based on observations of past behavior).

7 Preliminary results

The experiments we report in this paper are quite preliminary. While we are planning to run extensive simulations of the ASISA-CMP architecture, we do not yet have a fully-debugged simulation.

These experiments were all performed using Linux 2.6.18 kernels running on a dual-core Xeon model 5160 (3.0GHz, 64KB L1 caches, 4MB L2 cache).

7.1 Workloads

We used these workloads in our tests:

- **Netperf TCP_STREAM:** We used the TCP_STREAM benchmark from the Netperf [9] microbenchmark suite developed at Hewlett-Packard. In this benchmark, the client (the system under test) connects to a server and sends data as quickly as possible over a single TCP connection. It sets up a socket and calls `send()` in a tight loop. Normally this call returns as soon as the data is copied out of the user's buffer. However, if the kernel socket buffer is full, the call will block until space is available, potentially idling the processor. The user time spent executing this benchmark is minimal, and most of the CPU time is spent in the kernel driver managing the NIC or processing the packet in the TCP/IP stack. We used netperf's `--enable-dirty` configuration option, which ensures that its I/O buffers are always dirty in the data cache.

  We also used these netperf tests:
  - **TCP_MAERTS:** Like TCP_STREAM, but the server sends the data to the client
  - **TCP_RR:** Instead of streaming bulk data in one direction, in this test the client repeatedly executes request-response transactions with the server.
  - **TCP_CRR:** Like TCP_RR, but initiates a new connection for each transaction. (Note that we do not currently switch cores on the **connect** system call.)
- **Compile:** compile the Linux kernel, with optimization
- **Re-Aim-7 [25]:** an Open Source implementation of the AIM Benchmark Suite VII; AIM-VII is widely used by UNIX system vendors.

7.2 Time spent per system call

We ran a preliminary experiment to determine how much time is spent in the top half of each system call. In particular, for a given workload and a given system call, what is the average time spent in the call, and what fraction of time is spent in that call during the whole workload?

We modified the Linux 2.6.18 kernel (not the core-switching version of the kernel) to record the number of cycles spent in, and invocations of, each system call, not counting time when the process was blocked. For example, for a single minute-long **Web** trial, Table 4 shows the top 10 system calls, by total number of cycles, and excluding any calls that averaged under 2K cycles/call.

This is obviously a crude measure of which system calls are "expensive" enough to merit switching CPUs. The results will vary between architectures, kernel versions, applications, phases within an application, etc.

Table 4: System call costs: 1-minute **Web** trial

Call name	**Count**	**Tot. cyc.**	**Cyc./call**	**Cum. %**
sendfile	50627	3058.86M	60419	20.50%
stat64	78851	2226.89M	28241	35.42%
write	78473	2142.91M	27307	49.78%
writev	60164	1585.92M	26359	60.40%
socketcall	181108	1522.33M	8405	70.60%
read	141616	1309.60M	9247	79.38%
open	43165	1043.67M	24178	86.37%
poll	79373	743.66M	9369	91.36%
lstat64	90105	565.02M	6270	95.14%
close	49221	446.84M	9078	98.14%

As we described in Sec. 5.2, the decision in our modified kernel about whether or not to switch cores during a system call is, in some cases, conditional on how much work the call expects to do. Therefore, while Table 4 presents averages for cycles/call over all invocations of a given system call, our code switch on a subset of those calls with a higher mean.

7.3 Performance vs. power: methodology

Since we do not yet have the ability to simulate an ASISA-CMP system with a low-power OS-friendly core, we instead ran our modified Linux kernel on the dual-core Xeon system. We added instrumentation to this kernel to track when each core entered or exited the idle thread. We record the individual idle-time durations in a logarithmic histogram, with one bucket per power of two. This means that we can potentially underestimate the idle time by a factor of up to two, although the mean error should be smaller.

These idle time measurements allow to crudely model the performance vs. power effects of ASISA-CMP. Our current model depends on several big assumptions:

1. We can model the power of the application core by assuming it is approximately one third of the Thermal Design Power (TDP) of the CPU in our system. (We assume that the L2 cache consumes about as much power as either of the two cores.)
2. We can model the power and performance of the OS core by arbitrarily picking a plausible power level for a simplified core, and then assuming that OS code runs as fast on this core as it does on the actual CPU. This is clearly a bogus assumption, but it is offset somewhat by the excessive core-switching costs of our current implementation.
3. We can ignore the time it takes to transition a core out of a powered-down state, by assuming that this is smaller than the excess in core-switching costs. However, we do attempt to model the power consumed during these transitions.
4. We assume that the time to transition a core into a powered-down state is negligible.
5. Following Kumar *et al.* [13], we assume that powered-down cores "suffer no static leakage or dynamic switching power."
6. The timing of I/O events on the system we tested would not change significantly on an actual ASISA-CMP system, and hence we do not have to correct for differences in I/O waiting time.

Given these assumptions, our model has the following parameters:

- **Application core power:** 27 W (approximately 1/3 of the 80W TDP for the Intel 5160 chip).
- **OS core power:** we modelled several values of $P_{os} =$ 1W, 5W, and 10W.
- **CPU power-up time:** Following Kumar *et al.* [13], we assume that it takes 1000 cycles (333 nsec. at 3.0GHz) to transition out of a powered-down state. This is based partly on their assumption that a single phase-locked loop is shared among all cores, so the power-up delay is dominated by the time to charge and stabilize power buses.

7.4 Results

Table 5 shows results for the re-aim-7 benchmark. Results shown are the means of five trials. We ran two different test suites, "alltests" and "high_system", each with 5 and 20 simulated users. The high_system test apparently was designed to spend a lot of time executing in the operating system.

The table columns show the jobs per minute rate achieved (this is the re-aim-7 figure of merit); the amount of idle time we measured for each core in ASISA-CMP mode; and the jobs/minute/joule result for (respectively) the single-code system, and ASISA-CMP kernels when modelling $P_{OS} =$ 1W, 5W, and 10W, respectively. In this (admittedly crude) set of experiments, ASISA-CMP delivers a better jobs/minute/joule result than the single-core system, except when the ASISA-CMP OS core consumes 10W or more.

Table 5: Results from re-aim-7 benchmark

	Jobs/minute		ASISA-mode idle time		Jobs/minute/joule			
Configuration	1-core	ASISA	OS core	App core	1-core	ASISA/1W	ASISA/5W	ASISA/10W
alltests/5 users	2738	3091	8.22	5.93	7.88	18.44	14.41	11.32
alltests/20 users	7468	7598	12.53	3.53	15.50	19.14	16.26	13.68
high_system/5 users	1501	1498	16.33	15.45	2.43	6.68	4.74	3.48
high_system/20 users	5552	5489	16.82	14.04	8.37	17.39	13.23	10.19

Results are means for 5 trials

Table 6: Results from Netperf benchmark

	Bandwidth Mbit/sec		ASISA-mode idle time		Mbits/sec/joule			
Configuration	1-core	ASISA	OS core	App core	1-core	ASISA/1W	ASISA/5W	ASISA/10W
TCP_STREAM	941.50	941.37	43.56	37.34	0.58	1.40	1.03	0.78
TCP_MAERTS	941.38	941.42	43.68	40.20	0.58	1.58	1.13	0.83

	Transactions/sec		ASISA-mode idle time		Transactions/sec/joule			
Configuration	1-core	ASISA	OS core	App core	1-core	ASISA/1W	ASISA/5W	ASISA/10W
TCP_RR	11312.40	11329.90	45.86	37.95	6.55	14.76	11.07	8.43
TCP_CRR	4895.40	4877.19	47.44	37.49	2.83	6.25	4.71	3.60

Results are means for 5 trials

Results in Table 6 for the Netperf benchmarks show that ASISA-CMP, with all three values of P_{OS}, delivers better bandwidth or transactions per joule than the single-core system. (We ran 1-minute trials for these benchmarks, over a 1Gbit/sec Ethernet.)

Results in Table 7 for the Linux compilation, on the other hand, did not show a net benefit for ASISA-CMP. Even with $P_{OS} = 1$W, the ASISA-CMP system consumed more power during the compilation benchmark. Apparently, this benchmark mostly does relatively short system calls.

8 Summary

We have taken the first steps toward evolving Linux to support ASISA-CMP, and the first steps to evaluating its performance. The approach has promise; we need to do a more thorough study to prove it.

References

[1] J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. http://josh.trancesoftware.com/linux/, Feb. 2005.

[2] T. Agerwala. Computer Architecture: Challenges and Opportunities for the Next Decade. In *Keynote, ISCA 2005*, 2005.

[3] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI Throttling. In *Proc. ISCA*, pages 298–309, 2005.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. SOSP*, pages 164–177, 2003.

[5] T. Brecht, G. J. Janakiraman, B. Lynn, V. Saletore, and Y. Turner. Evaluating network processing efficiency with processor partitioning and asynchronous I/O. In *Proc. EuroSys*, pages 265–278, Leuven, Belgium, 2006.

[6] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proc. ISCA*, pages 13–23, San Diego, CA, June 2007.

[7] N. Gloy, C. Young, J. B. Chen, and M. D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proc. ISCA*, pages 12–21, 1996.

[8] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of Both Latency and Throughput. In *Proc. Int'l Conf. Computer Design*, pages 236–243, 2004.

[9] Hewlett-Packard Company. Netperf: A network performance benchmark. http://www.netperf.org.

[10] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., and Toshiba Corp. Advanced Configuration and Power Interface Specification.

Table 7: Results from Linux kernel compilation benchmark

Elapsed time		ASISA-mode idle time		joules			
1-core	**ASISA**	**OS core**	**App core**	**1-core**	**ASISA/1W**	**ASISA/5W**	**ASISA/10W**
312	319	225.12	5.21	8412	8797	10073	11669

Results are means for 5 trials

http://www.acpi.info/spec.htm, 2006.

[11] Koushik Chakraborty and Philip M. Wells and Gurindar S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *Proc. ASPLOS-XII*, San Jose, CA, Nov. 2006.

[12] T. Krazit. Intel pledges 80 cores in five years. *News.com*, Sept. 26 2006. http://news.com.com/Intel+pledges+80 +cores+in+five+years/2100-1006_3-6119618.html.

[13] 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-36*, San Diego, CA, Dec 2003.

[14] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. In *IEEE Computer*, 2005.

[15] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-Architecting VMMs for Multi-core Systems: The Sidecore Approach. In *Proc. Workshop on Interaction between Op. Systems and Comp. Arch.*, San Diego, CA, June 2007.

[16] K. Leigh. *Design and Analysis of Network and IO Consolidations in a General-Purpose Infrastructure*. PhD thesis, University of Houston, 2007.

[17] A. Mahesri and V. Vardhan. Power Consumption Breakdown on a Modern Laptop. In *Proc. Workshop on Power-Aware Comp. Sys.*, Portland, OR, Dec. 2004.

[18] D. McAuley and R. Neugebauer. A case for virtual channel processors. In *Proc. SIGCOMM Workshop on Network-I/O Convergence*, pages 237–242, Aug. 2003.

[19] D. Nellans, R. Balasubramonian, and E. Brunvand. A Case for Increased Operating System Support in Chip Multi-Processors. In *Proc.* $P = ac^2$ *Conf.*, Yorktown Heights, NY, Sept. 2005.

[20] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *USENIX Summer*, pages 247–256, Anaheim, CA, June 1990.

[21] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. In *Proc. ISCA*, pages 66–77, Boston, MA, June 2006.

[22] J. Rattner. Multi-core to the masses. In *Keynote, PACT 2005*, 2005.

[23] J. Redstone, S. J. Eggers, and H. M. Levy. An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In *Proc. ASPLOS*, pages 245–256, Cambridge, MA, Nov. 2000.

[24] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. *IEEE Computer*, 37(11):48–58, 2004.

[25] C. White and G. Payer. re-aim-7. http://sourceforge.net/projects/re-aim-7.

[26] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor Hardware Counter Statistics as a First-Class System Resource. In *Proc. HotOS-11*, San Diego, CA, May 2007.