
USING ASYMMETRIC SINGLE-ISA CMPS TO SAVE ENERGY ON OPERATING SYSTEMS

CPU'S 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 INTENDED TO IMPROVE APPLICATION PERFORMANCE. WE ADVOCATE ASYMMETRIC SINGLE-ISA MULTICORE SYSTEMS, IN WHICH SOME CORES ARE OPTIMIZED TO RUN OS CODE AT GREATLY IMPROVED ENERGY EFFICIENCY.

..... Our computer systems need to do better at balancing useful performance with energy consumption. In a recent paper, Barroso and Hölzle¹ pointed out that most servers operate most of the time at relatively modest utilizations—seldom entirely idle, and seldom fully utilized. However, typical servers consume almost as much energy when idle as they do when fully loaded, so their energy efficiency (useful work per joule) suffers in their normal operating region. Barroso and Hölzle argued that designers should “develop machines that consume energy in proportion to the amount of work performed.”

We know that many applications spend much of their execution in operating system (OS) code. (This has been documented by others,^{2,3} and we list our own measurements for several benchmarks later in this article, in Table 2.) But we have also known, since 1990, what John Ousterhout observed, that “operating system performance does not seem to be improving at the same rate as the

base speed of the underlying hardware.”⁴ So, OS code execution on complex CPU cores wastes energy, because it does not fully exploit that complexity.

Multicore systems offer us a way to restore energy proportionality to OS code execution, and more generally to OS-intensive applications, via asymmetric (or heterogeneous) cores. In 2003, Kumar et al. showed, through simulation, that energy efficiency could be improved by migrating application code between complex and simple cores as appropriate to the current phase of the application;⁵ their simulations did not include OS code. Because they migrated code dynamically, their proposal retained the single instruction set architecture (single-ISA) model of symmetrical chip multiprocessors: all cores can execute the same machine code. For conciseness, we refer to these as asymmetric single-ISA chip multiprocessor (ASISA-CMP) systems.

We advocate the use of ASISA-CMP systems in which some cores are optimized

Jeffrey C. Mogul
Jayaram Mudigonda
Nathan Binkert
Parthasarathy
Ranganathan
Vanish Talwar
HP Labs

for energy-efficient execution of OS code. These cores need not be novel designs; Nellans et al. 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].”² Older, simpler core designs, when implemented in the same VLSI process as modern complex cores, can consume an order of magnitude less power without a similarly large gap in OS performance.

Others have proposed heterogeneous multicore systems focused on OS code,^{2,6} but with multiple ISAs; that is, some cores run special ISAs, either dedicated to OS functions or dedicated to network processing. However, the nice thing about single-ISA heterogeneous multicore systems is that they eliminate worries about where the code executes, because any core can execute any part of the code. The single-ISA model makes it very easy to implement and maintain the necessary OS changes. The same kernel binary can execute on ASISA-CMPs and traditional, symmetric CMPs, with only minor configuration changes to optimize for energy efficiency.

Proportionality in energy consumption

There are at least two ways to design a system to meet the goal of energy proportionality: The first way is to use individual components whose power consumption varies with throughput, such as a CPU that supports voltage and frequency scaling. The second is to use a combination 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 ASISA-CMP model of Kumar et al.⁵ follows the second approach, without precluding the first one. Under light load, activity shifts to low-power cores; under heavy load, low-power cores can offer additional parallelism without significant increases in power consumption.

OS-friendly cores

We extend the ASISA-CMP model by asserting that low-power cores should be specialized to execute operating system

code. This assertion stems from several observations:

- OS code does not proportionately benefit from the potential speedup of complex, high-frequency cores. Running OS code on simpler cores is a better use of power and chip area.
- Most computer systems (with exceptions, such as those used for scientific computing) are often idle. (Ranganathan et al. report 90th-percentile utilization from nine different enterprise clusters between 5.3 percent and 44 percent;⁷ a subset of Google servers operates “most of the time” between 10 percent and 50 percent utilization.¹) Powering down complex CPU cores when they would otherwise be idle improves proportionality.

These observations led us to an approach based on two complementary designs:

- CMPs combining high-power, high-complexity application cores, and low-power, low-complexity OS-friendly cores; and
- operating system modifications to dynamically shift load to the most appropriate core, and (potentially) to power down idle cores.

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, buses, 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 the CPU represents the largest single power draw in a system, improving its proportionality is worthwhile. (For example, Mahesri and Vardhan report that a laptop’s CPU consumes up to 52 percent of total power.⁸)

ASISA-CMP offers the option of powering down high-power application core while maintaining OS functions on a low-power core. For example, the arrival of a new Web (HTTP/TCP) connection normally pre-

Table 1. Power and relative performance of Alpha cores at 2.1 GHz.*

Alpha core	Peak power (W)	Average power (W)	IPC**	Area**	Power**
EV4	4.97	3.73	1.00	1.00	1.00
EV5	9.83	6.88	1.30	1.76	1.84
EV6	17.8	10.68	1.87	8.54	2.86
EV8	92.88	46.44	2.14	82.2	12.45

* All cores scaled to 0.1 μm , at 2.1 GHz; IPC based on SPEC CPU benchmarks; this table is based on data from Kumar et al.⁵

** Normalized versus EV4.

cedes the arrival of the actual HTTP request by at least one network round-trip time (typically on the order of milliseconds or more). This delay would allow an OS core to handle the initial TCP connection request, letting the application core (if otherwise idle) continue sleeping until the actual HTTP request arrives.

Our approach naturally scales to a system with multiple application cores and multiple OS cores, in which individual cores are turned on as needed to handle the offered application or OS loads.

Core heterogeneity

The promise of ASISA-CMP depends critically on two facts of CPU core design:

- For a given process technology, a complex core consumes far more power and die area than a simple core.

- 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 performance per mm^2 . In fact, these performance results were based on the SPEC CPU benchmark suite; because operating system performance generally scales worse than application performance, the IPC ratios would be even smaller for OS code.²

Prior support for our assertion that OS performance scales worse than application performance comes from Ousterhout⁴ and from Brown and Seltzer;⁹ these results are over a decade old, and should be revalidated. We did a simplistic validation, running both nbench¹⁰ (a Linux port of the BYTE-mark application-benchmark suite) and hbench-OS⁹ (a suite of OS microbenchmarks) on 13 Linux 2.6.* systems running on many versions of x86 processors.

Figure 1 shows how several hbench-OS microbenchmarks scale relative to nbench's floating-point index. The circles show the rate of process creations (hbench's "simple static" test); the boxes show the rate of signal handler invocations. Both data sets are normalized to the slowest rate in the data set.

The solid and dashed lines show linear regressions for the process-creations and signals-handled data sets, respectively. Both are well below the $y = x$ line, which

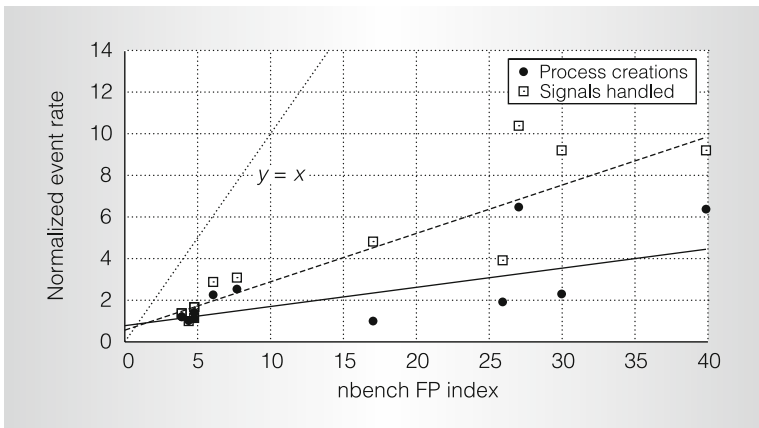


Figure 1. OS microbenchmark performance vs. floating-point performance. The solid line shows the linear regression for the process creations data set (circles); the dashed line shows the linear regression for the signals-handled data set (boxes).

represents equal scaling for OS and floating-point benchmarks. (Similar results hold when plotted against nbench's integer index.) This tends to confirm that OS performance scaling continues to lag behind application performance scaling. However, our simple experiment did not control for several factors—such as the precise Linux version or the details of the memory system—that might have skewed the results; there is room for a more careful study.

Complicating issues

Various issues complicate the question of whether our approach can improve throughput (the rate at which work is done) per watt. ASISA-CMP will pay off for OS code only if it reduces power consumption more quickly than it reduces performance.

ASISA-CMP can affect performance in several ways:

- *Running OS code on a slower CPU.* By design, ASISA-CMP may concede some performance by running OS code on a simpler core.
- *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 versus cache interference.* We assume that the cores in a CMP share one L2 cache but have private L1 caches. Core switching could harm cache affinity by requiring cache lines (for example, 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 far lower than for an application core (as discussed earlier), if there is available parallelism in the workload that extends to OS processing, an ASISA-CMP CPU could support more parallelism than a symmetric CMP 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. Clearly, not all workloads will have this kind of parallelism.

Competing approaches

There are definitely other approaches to improving throughput per watt in multicore systems, including several that are potentially complementary to ASISA-CMP:

- *Complex core with dynamic voltage and frequency scaling.* This could be especially effective at saving power in systems with a lot of idle time, but it might not reduce total core power as dramatically as turning off a complex core.
- *Complex core with power-down on idle,* with idle cores waking up on interrupts. This approach would not reduce the power consumed in OS code.

Several alternatives directly compete with ASISA-CMP, such as using a lot of (identical) simple cores, each of which can be powered off as needed. However, this configuration requires successful parallelization of application programs, a problem whose solution remains elusive. Others have proposed running the OS on cores with specialized ISAs,² but that would greatly increase the complexity of developing and maintaining an OS. (See the “Related Work” sidebar for more on other related approaches.)

Dynamic core-switching for OS kernel code

The ASISA-CMP approach could be applied to OS code in several ways:

- *Dynamically switching between cores in OS kernel code.* This technique is the subject of the rest of this article.
- *Binding device interrupts to OS-friendly cores.* Linux already offers this functionality; we enable it only for the network interface controller (NIC), since other devices interrupt far less often.

Related Work

Our work builds on that of Rakesh Kumar et al., the original proposal for using single-ISA heterogeneous multicore systems for saving energy.¹ However, their simulations did not include OS code. Since then, others have looked at the OS issue.

Nellans et al. measured the fraction of cycles spent in the OS for a variety of applications, and reexamined how OS performance scales with CPU performance, suggesting that interrupt-handling code interferes with caches and branch-prediction history tables.² 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. Wun and Crowley also proposed an asymmetric architecture, with specialized cores for running network code only.³

Grant and Afsahi looked at binding kernel threads to OS cores, to improve energy efficiency on scientific applications.⁴ They did not evaluate core switching during system calls, and their evaluation was based on throttling the clock of one core in a symmetric multiprocessor.

Chakraborty et al. proposed refactoring software so that similar *fragments* of code are executed on the same core of a CMP.⁵ Their initial study treated the OS and the user-mode application as two coarse-grained fragments, and they found speedup in some cases. However, they did not examine asymmetric CMPs or the question of power reduction.

Sanjay Kumar et al. proposed a *sidecore* architecture to support hypervisor operations.⁶ 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 the traps (for example, VMexit in Intel's VT architecture) that are used to invoke privileged hypervisor functions. Instead, the sidecore approach transfers the operation to a remote core "that is already in the appropriate state."⁶ This also avoids polluting the guest-core caches with code and data from hypervisor operations. (The sidecore approach does not specifically target saving energy.)

Superficially, asymmetric CMPs resemble the asymmetric multiprocessing (ASMP) of the 1970s. In that era, ASMP systems usually had identical CPUs. However, because of the difficulty of correctly managing parallelism within the operating system, these ASMP systems of necessity assigned specific tasks to specific processors. Most modern operating systems support symmetric multiprocessing (SMP); the ability to

balance load arbitrarily over the CPUs required a lot of hard work on OS implementation, but that work is now mostly complete. ASISA-CMP exploits SMP support in the kernel so that load can be moved to the most appropriate type of core. Thus, while ASMP solved the difficulties of implementing parallelism on identical CPUs, ASISA-CMP allows optimizing efficiency without requiring extensive work on kernel software.

References

1. R. Kumar et al., "Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction," *Proc. IEEE/ACM Int'l Symp. Microarchitecture (MICRO 03)*, IEEE CS Press, 2003, pp. 81-92.
2. D. Nellans, R. Balasubramonian, and E. Brunvand, "A Case for Increased Operating System Support in Chip Multiprocessors," *Proc. IBM Watson Conf. Interaction between Architecture, Circuits, and Compilers (P=ac² 05)*, 2005; <http://www.cs.utah.edu/~rajeev/pubs/pac205.pdf>.
3. B. Wun and P. Crowley, "Network I/O Acceleration in Heterogeneous Multicore Processors," *Proc. 14th IEEE Symp. High-Performance Interconnects (HOTI 06)*, IEEE CS Press, 2006, pp. 9-14.
4. R.E. Grant and A. Afsahi, "Power-Performance Efficiency of Asymmetric Multiprocessors for Multi-threaded Scientific Applications," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 06)*, IEEE CS Press, 2006, p. 344.
5. K. Chakraborty, P.M. Wells, and G.S. Sohi, "Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-Fly," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 06)*, ACM Press, 2006, pp. 283-292.
6. S. Kumar et al., "Re-architecting VMMs for Multi-core Systems: The Sidecore Approach," *Proc. Workshop on Interaction between Operating Systems and Computer Architecture (WIOSCA 07)*, 2007; <http://www.ideal.ece.ufl.edu/workshops/wiosca07/Paper3.pdf>.

- *Binding kernel threads to OS-friendly cores.* Grant and Afsahi have shown that this improves energy efficiency on multi-threaded scientific applications.¹¹ However, such relatively static binding might make load balancing more difficult.
- *Running virtualization "helper processing" on OS-friendly cores.* Systems such as Xen¹² use a privileged virtual machine ("Domain 0") to multiplex and manage I/O operations for other virtual machines. This code could run on an OS-friendly core.
- *Running "OS-like code" on OS-friendly cores.* The definition of an operating system is quite fluid; the same code that executes inside the kernel in a monolithic system would execute in user mode on a microkernel. Also, some applications, such as Web servers, have instruction-level behavior similar to OS code. The choice of whether to execute code on OS-friendly cores should not depend simply on whether it runs in kernel mode.

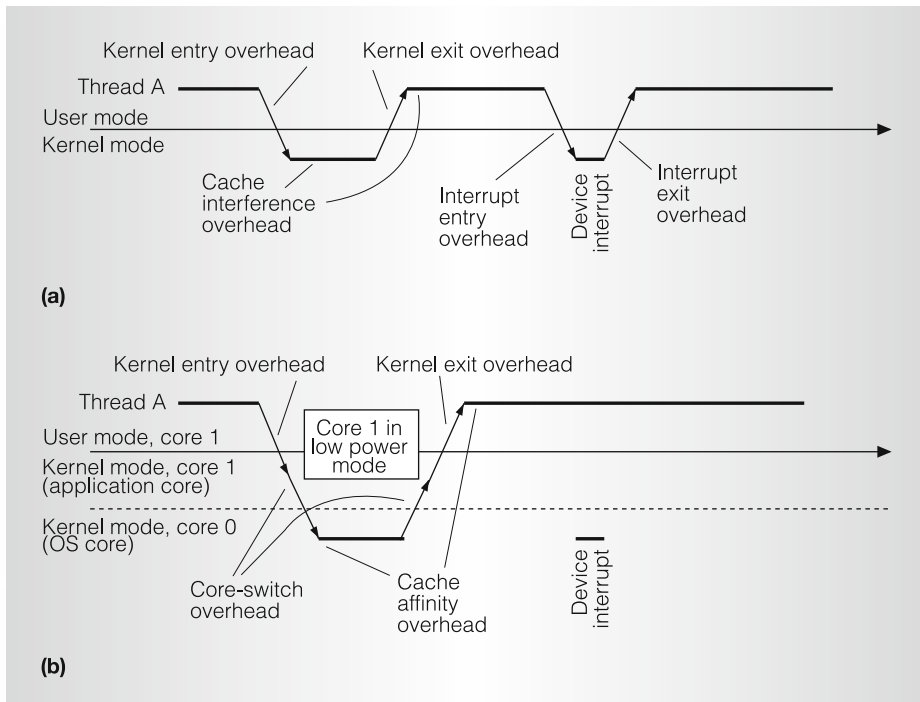


Figure 2. Example timeline showing overheads, without thread-level parallelism: single-core CPU (a) and asymmetric single-ISA chip multiprocessor (ASISA-CMP) dual-core CPU (b).

To date, we have focused all of our implementation and experiments on dynamic core switching for OS kernel code (with interrupt binding enabled). In an ASISA-CMP system, there are two ways to optimize throughput per watt:

- If the system is underutilized, shift OS load to low-power cores and power down high-power cores whenever possible.
- 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 in which no application-level parallelism is available. Figure 2 illustrates a simple example for an application with just one thread. Figure 2a 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 2b 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 transfers control from the “application core” (core 1) to the “OS core” (core 0); puts core 1 in low-power mode; executes the system call on core 0; wakes up core 1; and transfers control back to core 1.

Figure 2a and Figure 2b also show what happens on the arrival of an interrupt. In the single-core system, application execution is delayed 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.

Next, consider the case in which application-level parallelism exists. Figure 3 (next page) illustrates another simple example, for an application with two runnable threads, A and B, where thread A is running at the start of the example, and again the

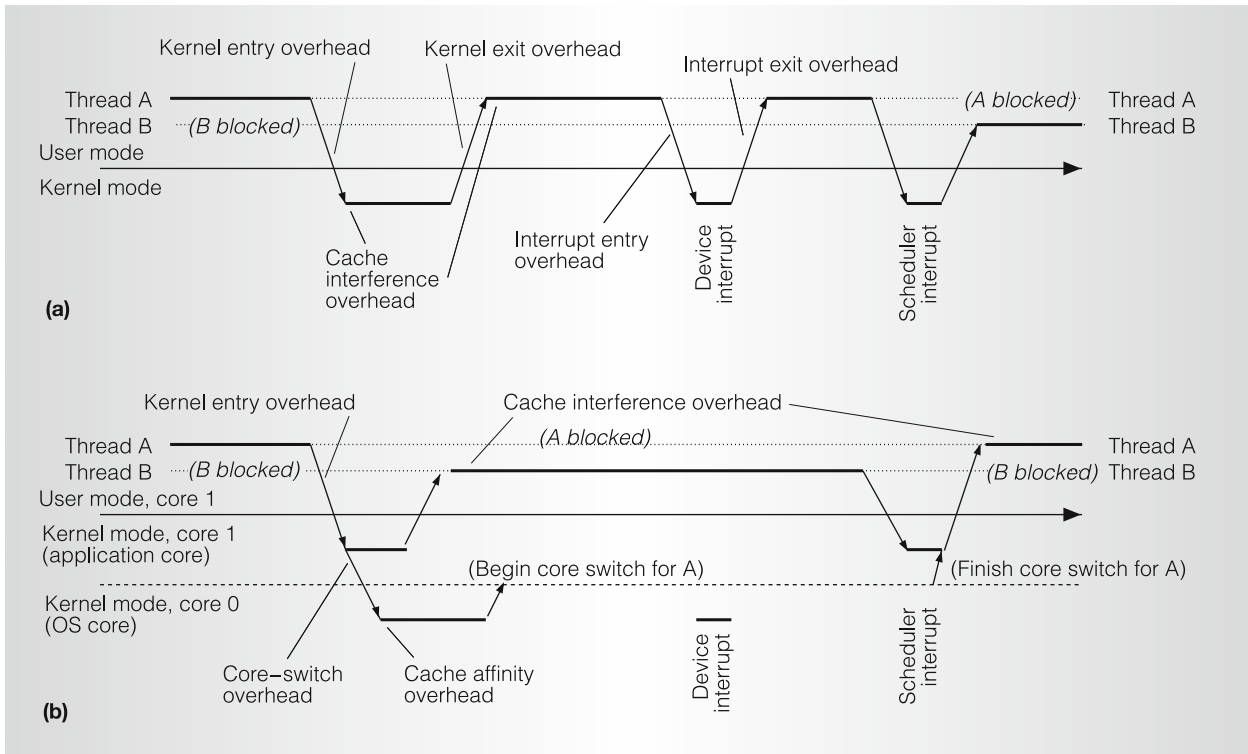


Figure 3. Example timeline with thread-level parallelism: single-core CPU (a) and ASISA-CMP dual-core CPU (b).

thread makes a system call. Figure 3a 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 3b 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).

Obviously, dynamic core switching is useful only if the benefits of running OS code on specialized cores—either the ability to shut down high-power cores or the ability to get more parallelism from those cores—outweigh the switching overheads and the reduced performance of the simpler cores.

Implementation of dynamic core switching

We implemented dynamic core switching in Linux (version 2.6.18); our changes were relatively simple. Our implementation addressed the two main challenges: deciding when to switch, and switching quickly.

Switching cores takes time because it adds extra instructions, it requires transferring some state between CPUs, and if the

target core is powered down, it could take about a thousand cycles to restart it.⁵ The trade-off is only beneficial for expensive, frequent system calls (for example, **select** or perhaps **open**), not fast or rare calls (such as **getpid** or **exit**). For some system calls, the decision to switch also depends on how much work is to be done. A **read** system call with a 10-byte buffer should not switch; a 10-Kbyte read probably should.

We modified certain system calls so that the invocation sequence is as follows (the steps in boldface are our modifications):

1. Validate arguments.
2. **Decide whether there is enough work to do to merit switching.**
3. **If so, invoke a core-switching function (`asisaSwitchCores`) to switch the thread to an OS core.**
4. Do the bulk of the system call.
5. **If the thread switched cores, switch back to an application core.** Return to the original core, if possible, to preserve cache affinity.
6. Finish up and return from the system call.

The details differ slightly for each system call, but generally involve only a few lines of new code.

In our current implementation, we switch cores during these system calls: **open**, **stat**, **read**, **write**, **readv**, **writew**, **select**, **poll**, and **sendfile**. For **read**, **write**, and similar calls, we arbitrarily defined “enough work” as 4,096 bytes or more; we have not yet tried to optimize this choice.

We dedicate one or more cores to execute only OS code; we modified the kernel to maintain bitmaps designating the OS cores and the application cores. Our **asisa_SwitchCores** function sets a new per-thread field specifying a target CPU, then invokes the Linux scheduler (**schedule**), the subject of our most extensive changes.

Our modified **asisa_schedule** checks the target CPU field. If the current thread wants to switch, **asisa_schedule** remembers this, deactivates the thread, then finds a new process to run on the current CPU as usual. After the context switch, if the original thread wants to switch cores, **asisa_schedule** moves that thread to the target CPU’s run queue, then sends (if necessary) the “reschedule” inter-processor interrupt to that CPU. This interrupt causes **schedule** to be invoked on the destination CPU, which then activates the newly arrived thread (the unmodified behavior of **schedule**). Our design respects the existing Linux thread-priority scheme.

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. (In practice, one would never actually want to switch cores during **getpid**.) 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.

On a dual-core Xeon model 5160 (3.0 GHz, 64-Kbyte L1 caches, 4-Mbyte L2 cache), **getpid** takes about 84 ns without core switching enabled, and about

3,400 ns with core switching enabled—a slowdown of about 4,000 percent. However, when we express the overhead as a ratio, **getpid** provides the worst-case comparison. In contrast, the **read** system call, with a 64-Kbyte buffer and reading from a warm buffer cache, takes about 11.5 μ s without core switching, and about 15.0 μ s with core switching, an increase of only 31 percent. If the file is not in the buffer cache, the **read** would take several milliseconds, and the overhead from core switching would be negligible.

Open issues

So far, our work on dynamic core switching only partly solves the scheduling problems associated with asymmetric cores. Many issues remain unsolved, especially in finding a balance between energy efficiency and high throughput. For example, our current scheduler is not work-conserving: it may leave an application core idle even when the OS core is overloaded. We also have not addressed the problem of efficiently distributing load across multiple OS cores, or how our changes interact with Linux’s load balancer. Finally, we have not systematically evaluated which system calls merit switching, or what the right buffer size thresholds are.

Design issues for OS processors

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

Caches

OS code tends to have higher cache miss rates than application code.³ This suggests that the per-core first-level (L1) caches for OS cores might benefit from different parameters (line size, associativity, total cache size, and so on) than those for application cores. An OS core could have smaller caches overall, or it could have a larger L1 instruction cache at the expense of a smaller L1 data cache.

Pipeline

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 far more complex pipeline, but with significant power and area savings.

Branch-prediction tables

Although we know of no studies addressing whether kernel-only execution would be optimized by a different table size than that for mixed kernel + user execution, Gloy et al. reported that simulations of branch prediction should not rely on user-only traces if the kernel accounts for even 5 percent of execution time.¹³ This result suggests that an OS-friendly core might need a different branch-prediction 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

Although this article focuses on CMPs with a single OS-friendly core, there is no inherent reason why there should be just one. Future CMPs might have dozens of cores, so an ASISA-CMP designer would probably have to choose the appropriate fraction of OS-friendly cores to optimize power versus performance for a range of anticipated workloads. (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 toward on-chip integration of I/O hardware (for example, PCI Express controllers, HyperTransport, or Intel's QuickPath). Although 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, thereby improving performance and reducing power consumption.

Preliminary experiments

The experiments we report here are quite preliminary. Our goal was not to demonstrate the best possible results or how to get them, but rather to show that ASISA-CMP provides at least some improvement. (Readers could, and should, object to almost every simplification we make here.) We will have to do more extensive experiments to demonstrate the value of ASISA-CMP and to answer the open design questions. In particular, we have tested just a few arbitrary choices for core complexity and clock speed, rather than trying to find the optimal trade-offs, and we have not optimized our kernel's decision about when to switch cores.

Methodology

The key question is whether an ASISA-CMP can improve the performance/joule ratio for realistic workloads. We addressed this question using cycle-level simulation and simple models of power consumption.

We used M5,¹⁴ a multiprocessor simulator that does cycle-level (as opposed to behavioral) simulations and that supports real operating system code. We currently have M5 models only for the Alpha architecture (x86 models are in progress, but will not be running soon). The simulator is fully deterministic, so we ran only one trial for each experiment. (We are considering adding some nondeterminism for future experiments.)

We ran the following OS-intensive benchmarks: netperf's TCPstream and netperf's TCPmaerts (which differ in the direction that data flows) and Apache with a workload based on SPECweb. Table 2 shows that these applications spend a significant fraction of their non-idle time in kernel or interrupt modes, although at lower NIC speeds the netperf benchmarks often leave the CPU idle.

We booted the kernel and started the applications using a coarse-grained simulation, then switched to fine-grained simulation for the measurements we report in the "Performance versus power" section (later

in the article). Fine-grained simulations run at slowdowns of five or six orders of magnitude, and generate large trace files to support our energy modeling, so currently our fine-grained simulations cover about 167 ms per netperf test, and 133 ms per Web test.

Simulated configurations. We simulated the following system configurations: *uni*, with a single (uniprocessor) 3-GHz complex core; *duo*, with two 3-GHz complex cores; *mini*, with two 3-GHz simple cores; *scaling*, with two complex cores, one at 3 GHz and one (for the OS) at 750 MHz; *asisa 1*, with one complex core and one simple core, both at 3 GHz; and *asisa 2*, with one 3-GHz complex core and one 750-MHz simple core.

For each dual-processor hardware configuration, we ran tests using an unmodified Linux (*base*), an unmodified Linux that binds Ethernet and disk interrupts to the OS core but does not switch cores (*bound*), and our core-switching Linux, also with interrupts bound to the OS core (*new*).

Power consumption models. We followed the same power estimation methodology as Kumar et al.⁵ Given the asymmetric nature of our processor cores, the individual power consumption is a function of the circuit design style and process parameters, as well as processor configuration and workload. We assume that all cores run at 1.2 V and are scaled to the same 0.10-micron technology. (This might underestimate the energy improvements from asymmetry, because slower cores could run at lower voltages.) In this article, we focus on peak power; for most benchmarks, the ratio of typical power to peak power is fairly consistent across the two core configurations that we consider.⁵ Consequently, peak power provides a first-order estimate of available power savings. Table 3 shows the specific peak power values we used. The core power models from Kumar et al.⁵ are all existing Alpha cores, designed for high performance on application code, and therefore include application-quality floating-point units. In a real ASISA-CMP system, we might expect that the OS

Table 2. Fraction of CPU time spent in various modes.*

NIC speed (Gbps)	Benchmark	Percentage of time spent in mode			
		User	Kernel	Interrupt	Idle
1	Apache	55.7	33.1	9.0	2.2
1	TCPmaerts	29.4	28.2	8.5	33.9
1	TCPstream	0.0	8.2	8.9	82.9
10	Apache	55.7	32.8	9.0	2.4
10	TCPmaerts	60.7	29.7	7.7	2.0
10	TCPstream	0.1	26.0	17.4	56.5

* Measurements are based on unmodified Linux on a simulated uniprocessor.

cores would be redesigned specifically for efficient OS execution and might draw even less power than the simple cores in Table 3.

We simulated a shared L2 cache (3.5 Mbytes, 8-way set associative, 12-ns access), obtained access statistics from the simulator, and modeled the energy consumed per access: 28 nJ for a read hit, 32 nJ for a write miss, and 56 nJ for a read miss. (These cache-access energy estimates were provided to us by R. Kumar.) L2 cache accesses consumed about 2 percent to 23 percent of the total energy across our various tests, depending on the configuration.

Metrics. For each combination of workload and configuration, we measured the useful work done during the fine-grained simulation, and the total energy consumed. Our figure of merit, performance per joule, is the ratio of those measurements. For our networking benchmarks, we define *useful work* simply as the total number of network bytes sent and received.

Modeling core power-up

We follow Kumar et al.⁵ both in assuming that “unused cores are completely powered down, rather than left idle,” to

Table 3. Core peak-power models.

Core name	Alpha core type	Peak power (W) at 750 MHz	Peak power (W) at 3 GHz
Simple	EV4-like	1.78	7.1
Complex	EV6-like	6.4	25.4

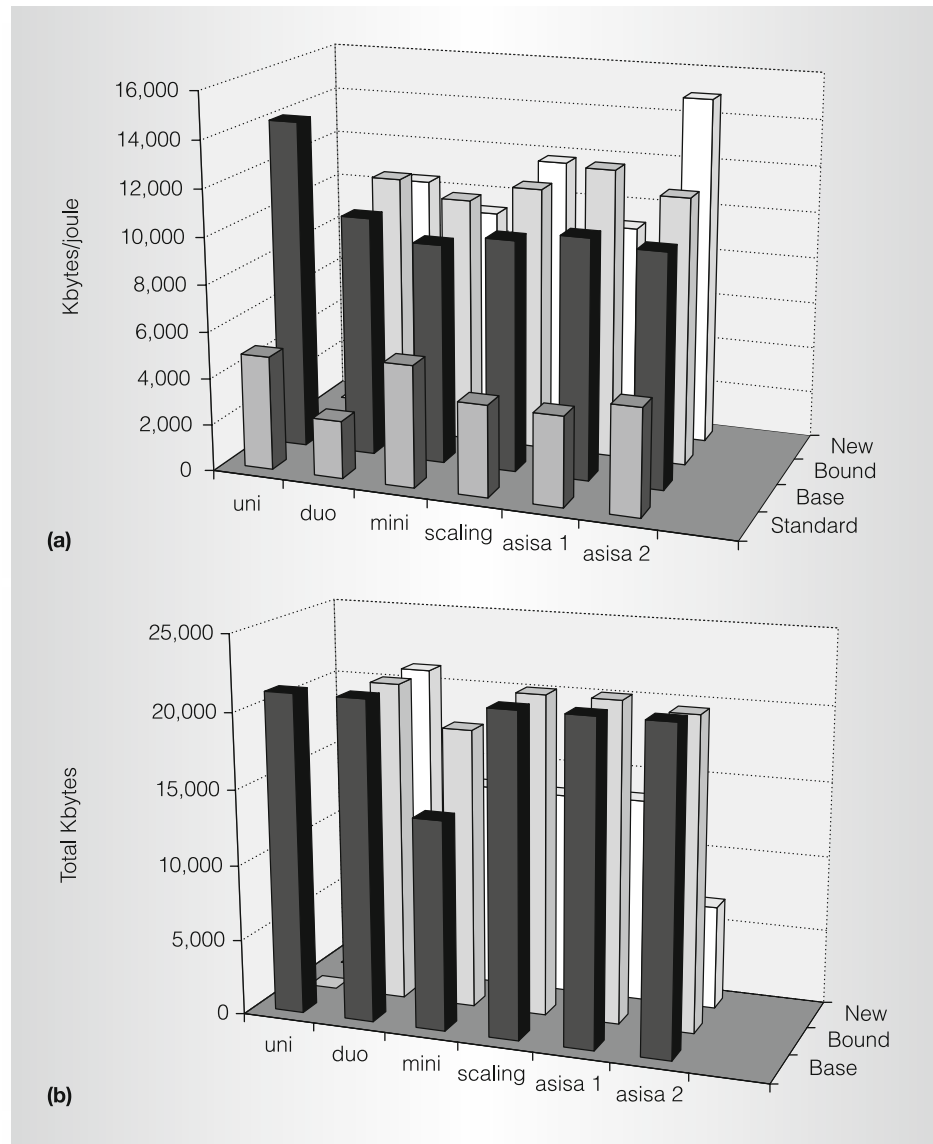


Figure 4. Performance versus power for TCPstream, with NIC bandwidth of 1 Gbps: Kbytes per joule (a) and total Kbytes (b).

minimize wasted power, and in estimating the power-up delay at about 1,000 cycles. This delay is independent of the clock frequency; it represents the time required to charge the core's power distribution network, assuming these wires are sized primarily to handle the core's normal operating currents.

We assume that powering down a core costs nothing, and that the latency of powering up a core can be fully overlapped with the somewhat longer latency of going through the scheduler. This latter assump-

tion is probably optimistic, but we currently have no way to measure it. However, we must model the energy cost of waiting for a powered-down core to restart. We do this by subtracting 1,000 cycles from the duration of each idle interval, then summing the nonnegative results to get the total idle time for each core.

We also have a choice: Should we assume that all idle cores are powered down, or only the application cores? And should we assume that an otherwise unmodified kernel could also power down idle cores? To be

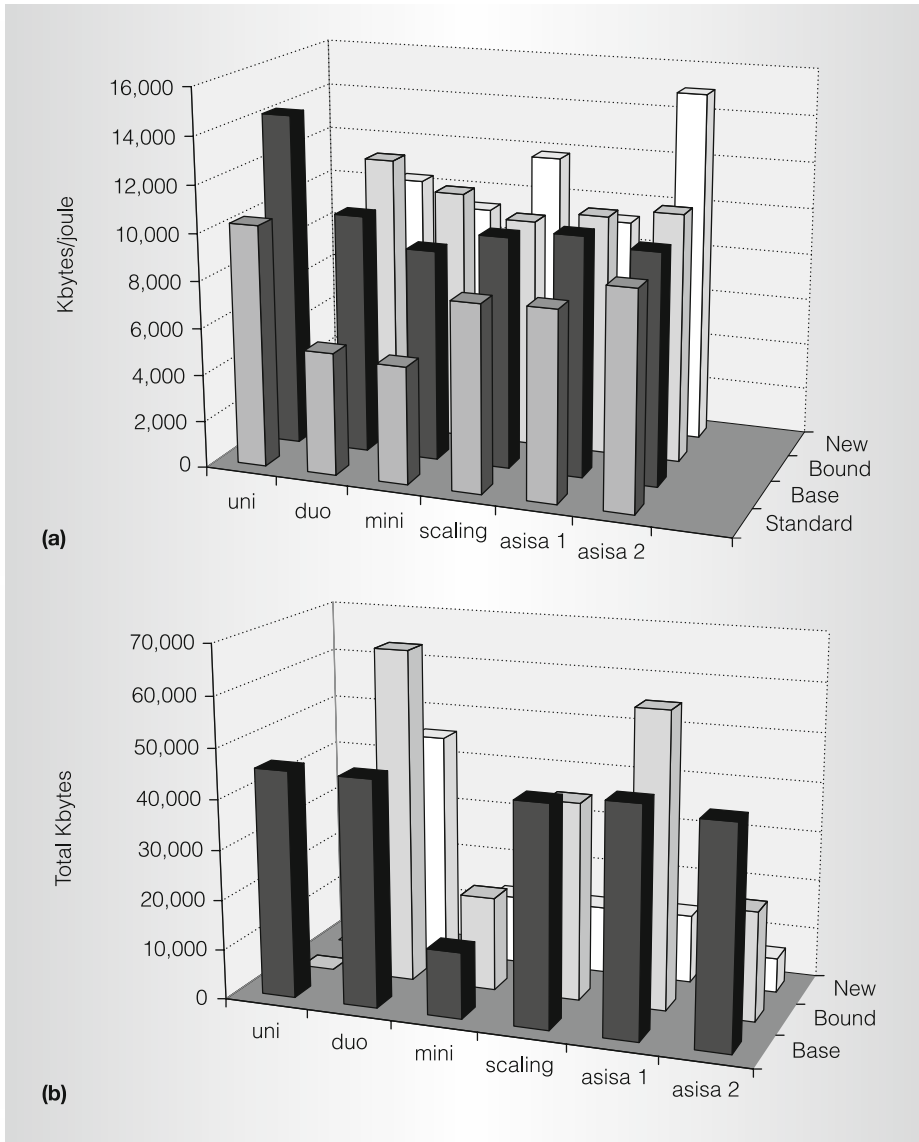


Figure 5. Performance versus power for TCPstream, with NIC bandwidth of 10 Gbps): Kbytes per joule (a) and total Kbytes (b).

conservative, our energy models assume that all kernels (including base) power down all idle cores; we also model a standard configuration in which the base kernel never powers down any core.

Performance versus power: Results

In the TCPstream benchmark, the system under test sends TCP data as fast as possible. For the *bound* and *new* software configurations, netperf was pinned to the application core.

Figures 4 and 5 show results for 1-Gbps and 10-Gbps Ethernet bandwidths. Generally, the uniprocessor system with the unmodified kernel provides the best efficiency. However, the *asisa 2* hardware with the *new* (core-switching) kernel gets somewhat better throughput per watt. The OS core is saturated in this configuration (as the total Kbytes results show), which limits total throughput. We suspect that a CMP with several OS cores could achieve higher bandwidths with similar efficiency.

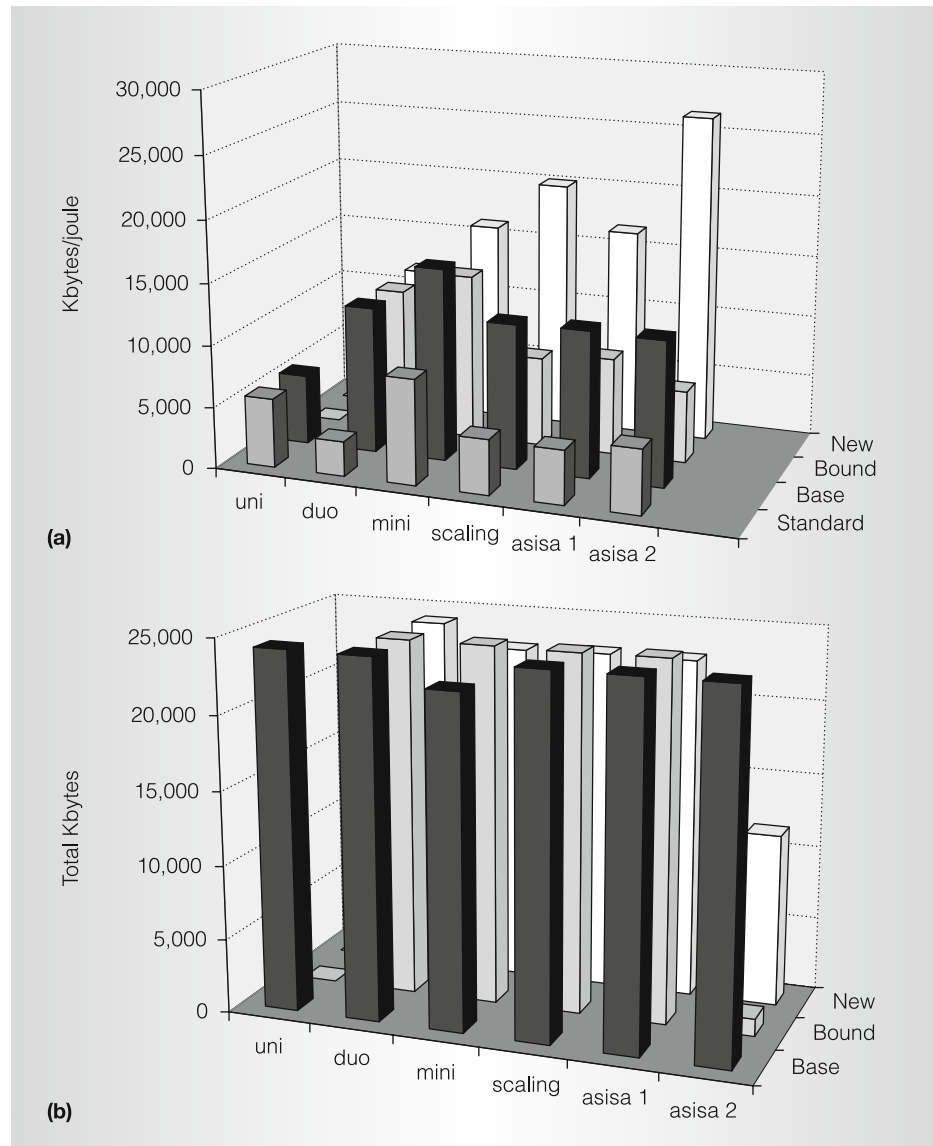


Figure 6. Performance versus power for TCPmaerts, with NIC bandwidth of 10 Gbps: Kbytes per joule (a) and total Kbytes (b).

In the TCPmaerts benchmark, the system receives (and discards) TCP data as fast as possible. Figure 6 shows results for 10-Gbps NIC bandwidth. The *new* (core-switching) kernel is significantly more energy-efficient on several different configurations; again, the *asisa 2* hardware seems underpowered and might support higher throughput with multiple OS cores.

The netperf benchmarks are single stream and do not get much multiprocessor speedup. However, because TCP acknowl-

edgment packet reception can run in parallel with TCP transmission, binding interrupts to the OS core can (but does not always) improve throughput and energy efficiency on TCPstream, compared to the base operating system configuration. Binding interrupts does not help TCPmaerts, in which most ACKs are sent as part of the interrupt-driven processing for received data packets, and hence there is no chance for parallelism between data processing and ACK processing.

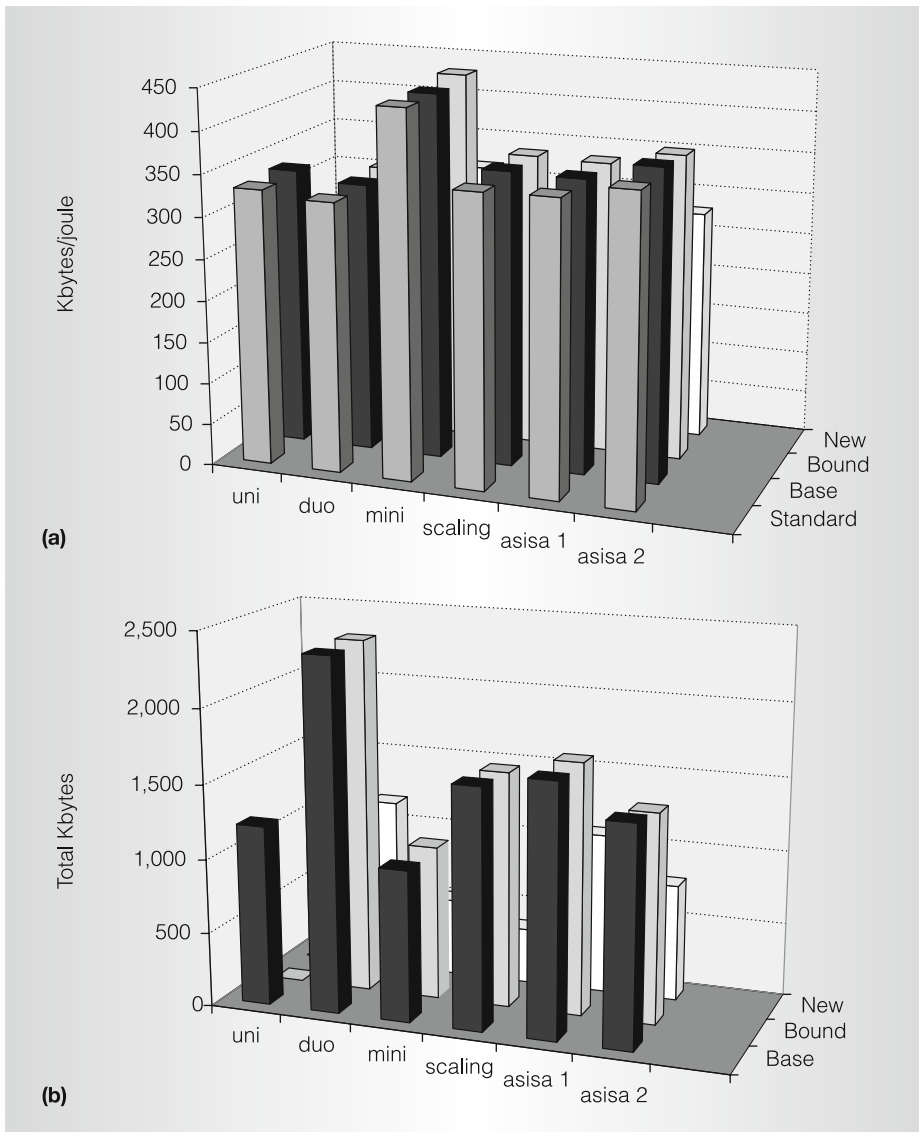


Figure 7. Performance versus power for Apache benchmark, with NIC bandwidth of 1 Gbps: Kbytes per joule (a) and total Kbytes (b).

Figure 7 shows results for Apache on 1-Gbps Ethernet. These tests are CPU limited, although the *new* kernel, which we ran with Apache pinned to the application core, saturates the application core but sometimes leaves the OS core idle. In the other tests, we left Apache unpinned.

At first, we were dismayed to see that the *mini* hardware without core switching consistently had the best energy efficiency. But then we realized that, because Apache’s code is “OS-like” as defined earlier (see “Dynamic core switching for OS kernel

core”), and the benchmark spends roughly half its time in user mode, a system with only OS-friendly cores is the best platform for this application, which is known to scale well on multiprocessors. Core switching would not be useful in that configuration.

Our preliminary experiments have shown that the ASISA-CMP approach can improve energy efficiency on applications that frequently execute operating system and OS-like code. Further research, including analysis of hardware

design choices and probably more software improvements, may lead to scalable CMP systems with better energy efficiency on real-world applications. MICRO

References

1. L.A. Barroso and U. Hözl, "The Case for Energy-Proportional Computing," *Computer*, vol. 40, no. 12, Dec. 2007, pp. 33-37.
2. D. Nellans, R. Balasubramonian, and E. Brunvand, "A Case for Increased Operating System Support in Chip Multi-processors," *Proc. IBM Watson Conf. Interaction between Architecture, Circuits, and Compilers (P=ac² 05)*, 2005, <http://www.cs.utah.edu/~rajeev/pubs/pac205.pdf>.
3. J. Redstone, S.J. Eggers, and H.M. Levy, "An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 00)*, ACM Press, 2000, pp. 245-256.
4. J.K. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proc. Usenix Summer 1990 Conf.*, Usenix Assoc., 1990, pp. 247-256.
5. R. Kumar et al., "Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction," *Proc. IEEE/ACM Int'l Symp. Microarchitecture (MICRO 03)*, IEEE CS Press, 2003, pp. 81-92.
6. B. Wun and P. Crowley, "Network I/O Acceleration in Heterogeneous Multicore Processors," *Proc. 14th IEEE Symp. High-Performance Interconnects (HOTI 06)*, IEEE CS Press, 2006, pp. 9-14.
7. P. Ranganathan, et al., "Ensemble-Level Power Management for Dense Blade Servers," *Proc. Int'l Symp. Computer Architecture (ISCA 06)*, IEEE CS Press, 2006, pp. 66-77.
8. A. Mahesri and V. Vardhan, "Power Consumption Breakdown on a Modern Laptop," *Proc. Workshop Power-Aware Computer Systems*, LNCS 3471, Springer-Verlag, 2005, pp. 165-180.
9. A.B. Brown and M.I. Seltzer, "Operating System Benchmarking in the Wake of Imbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," *Proc. Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS 97)*, ACM Press, 1997, pp. 214-224.
10. U.F. Mayer, "Linux/Unix nbench," 2004, <http://www.tux.org/~mayer/linux/bmark.html>.
11. R.E. Grant and A. Afsahi, "Power-Performance Efficiency of Asymmetric Multiprocessors for Multi-threaded Scientific Applications," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 06)*, IEEE CS Press, 2006, p. 344.
12. P. Barham et al., "Xen and the Art of Virtualization," *Proc. Symp. Operating Systems Principles (SOSP 03)*, ACM Press, 2003, pp. 164-177.
13. N. Gloy et al., "An Analysis of Dynamic Branch Prediction Schemes on System Workloads," *Proc. Int'l Symp. Computer Architecture (ISCA 96)*, ACM Press, 1996, pp. 12-21.
14. N.L. Binkert et al., "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, Jul.-Aug. 2006, pp. 52-60.

Jeffrey C. Mogul is a Fellow at HP Laboratories. His research interests include operating systems and computer networks. Mogul has a PhD in computer science from Stanford University. He is a Fellow of the ACM.

Jayaram Mudigonda is a research scientist at HP Laboratories. His research interests include networking, operating systems, and architecture. Mudigonda has a PhD in computer sciences from the University of Texas at Austin.

Nathan Binkert is a research scientist at HP Laboratories. His research interests lie at the intersection of computer architecture, operating systems, and networking. Binkert has a BSE in electrical engineering, and an MS and a PhD in computer science, all from the University of Michigan.

Parthasarathy Ranganathan is a principal research scientist at HP Laboratories. His research interests include low-power design, system architecture, and parallel computing. Ranganathan has a PhD in electrical and computer engineering from Rice University.

Vanish Talwar is a research scientist at HP Laboratories. His research interests include distributed systems, operating systems, and

computer networks, with a focus on management technologies. Talwar has a PhD in computer science from the University of Illinois at Urbana Champaign.

Direct questions and comments about this article to Jeffrey Mogul, 1501 Page Mill

Road, MS 1177, Palo Alto, CA 94304; jeff.mogul@hp.com.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/csdl>.

Lower nonmember rate of \$29 for *S&P* magazine!

IEEE Security & Privacy magazine is the premier magazine for security professionals. Each issue is packed with information about cybercrime, security & policy, privacy and legal issues, and intellectual property protection.

Top security professionals in the field share information you can rely on:

- Silver Bullet podcasts and interviews
- Intellectual Property Protection and Piracy
- Designing for Infrastructure Security
- Privacy Issues
- Legal Issues and Cybercrime
- Digital Rights Management
- The Security Profession

Subscribe now!

www.computer.org/services/nonmem/spbnr

