To appear in ACM SIGOPS Operating Systems Review special issue on
***The Interaction Among the OS, the Compiler, and Multicore Processors*, April, 2009, Volume 43, Number 2**

Fast Switching of Threads Between Cores

Richard Strong
UC San Diego
rstrong@cs.ucsd.edu

Jayaram Mudigonda
Jeffrey C. Mogul
Nathan Binkert
HP Labs
Jayaram.Mudigonda@hp.com
Jeff.Mogul@hp.com
Binkert@hp.com

Dean Tullsen
UC San Diego
tullsen@cs.ucsd.edu

Abstract

We address the software costs of switching threads between cores in a multicore processor. Fast core switching enables a variety of potential improvements, such as thread migration for thermal management, fine-grained load balancing, and exploiting asymmetric multicores, where performance asymmetry creates opportunities for more efficient resource utilization. Successful exploitation of these opportunities demands low core-switching costs. We describe our implementation of core switching in the Linux kernel, as well as software changes that can decrease switching costs. We use detailed simulations to evaluate several alternative implementations. We also explore how some simple architectural variations can reduce switching costs. We evaluate system efficiency using both real (but symmetric) hardware, and simulated asymmetric hardware, using both microbenchmarks and realistic applications.

1. Introduction

We are in the multicore era: CPU vendors currently offer as many as 16 cores per chip. One implication of the shift to multicore CPUs is that inter-core communication costs have dropped by orders of magnitude, compared to traditional multiprocessors. This drop creates the potential for significantly more nimble and dynamic management of executing threads, since it reduces the time for hardware to move a migrating thread's data working set (typically residing in a shared cache) by similar orders of magnitude. Potential improvements enabled by fast thread migration, or "core-switching", include thermal management [7, 13], fine-grained load-balancing [3], and exploiting asymmetric cores [20] by moving computation when execution characteristics change.

However, we cannot exploit these opportunities while the *software* costs of moving a thread remain as high as they are now, since these costs can dominate the *communication* costs of moving the working set. Our research seeks to understand and reduce these software costs.

We are particularly interested in the potential of asymmetric multicore processors, because core-to-core performance asymmetry appears to be a useful way to improve energy and area efficiency. An asymmetric multicore CPU has cores which, while they all execute more or less the same instruction set, vary greatly in complexity, and hence size and energy consumption. Code that has no need for a complex core can be run on a simpler core, often with relatively little performance cost but with greater throughput per watt.

The use of asymmetric multicore processors increases the need for frequent migration of threads between cores; the potential to gain efficiency from a core-switch can arise arbitrarily often.

Previous work has shown the potential value of multicore performance asymmetry, starting with Kumar *et al.* [20], who first proposed an asymmetric single-ISA (or "ASISA", for short) multicore. They used simulations to show that dynamic migration of application code between cores of varying complexity could improve energy efficiency. Nellans *et al.* [28] and Wun and Crowley [34] independently proposed asymmetric architectures with specialized cores, respectively, for operating system (OS) code and for network code. (However, Nellans and his colleagues now suggest that adding OS-specific caches to standard cores might be a better idea than using additional cores for OS execution [30].)

Balakrishnan *et al.* [2] point out that an asymmetric multiprocessor can outperform a symmetric processor "in which all cores are slow because the fast core is effective for serial portions" of a parallelized program. Grant and Afsahi [12] evaluated an ASISA approach in which specific kernel threads are bound to cores optimized for OS performance, using actual hardware and throttling the clock of one core to emulate the performance of a low-power core. In previous work [25], we evaluated the use of simpler cores to run OS (and "OS-like") code, using simulation to show the potential for improved energy efficiency.

Much of the prior work dynamically matches the characteristics of code to the appropriate core – that is, exploiting performance asymmetry requires switching cores during the execution of a thread. For example, Balakrishnan *et al.* showed that, to get good performance, the OS scheduler sometimes needs to migrate a thread from a slow core to a fast, idle core[2]. Thread migration is also necessary when balancing loads between cores (in a symmetric or asymmetric system). However, the benefits of dynamic migration must be balanced against the costs, especially if the benefits come from frequent migrations to exploit the performance characteristics of relatively short execution segments.

Therefore, core-switching costs are a fundamental factor when evaluating the feasibility of asymmetric multicore systems. As far as we are aware, however, no previous work has carefully evaluated the software costs of core-switching, or how these might be reduced.

In this paper, we make the following contributions:

- We cut Linux core-switching latencies in half or better, compared to previous mechanisms for thread migration.
- We explain the basis of our approach, as modifications to the existing Linux scheduler (sec. 3).
- We show several ways to reduce the cost of software-based core-switching (sec. 6).
- Using simulations, we explore how core-switching costs depend on several architectural parameters (sec. 7).

- We present both micro-benchmarks and macro-benchmarks evaluating the efficiency of core-switching, on both real and simulated hardware.

In addition:

- We describe some additional architectural changes that could further improve core-switching efficiency (sec. 6).
- We describe several different policies for deciding when to switch cores (sec. 9.2), although their evaluation remains as future work.

Our ultimate goal in this paper is to provide a sound basis for further work on multicore systems, especially but not exclusively asymmetric multicores, that depend on rapid dynamic migration of threads between cores.

1.1 Motivation

Hill and Marty recently pointed out [14] that asymmetric multicores are an inevitable consequence of scaling to large numbers of cores on a chip: by analogy to Amdahl's Law, highly-parallelizable applications will have sequential code that runs best on a core with the best possible single-thread performance.

Earlier, Kumar *et al.* [20] argued for asymmetric cores because some code gains little benefit from complex cores, and, if executed on a simple core, will run at nearly the same speed; simpler cores tend to consume much less energy per instruction executed. They observed that simple cores can be a better match for memory-bound application code. Previously, we observed that operating system and "OS-like" code is typically memory-bound, and hence should be a good match for energy-efficient execution on simple cores in asymmetric CPUs [25].

Kumar *et al.* [20] modeled core switching at a very coarse granularity, allowing them to ignore the cost of core-switching. However, the introduction of cores specialized for OS and similar code implies the potential for much more frequent migration, potentially making performance highly sensitive to that cost. Becchi and Crowley [3] modelled core switching for a similar architecture, using a somewhat more sophisticated model including the execution of an OS, but while they modelled the data-communication costs of core switching, they apparently did not try to model the kernel software overheads.

Other scenarios that could create frequent core-switching include: compiler-initiated or programmer-initiated thread migrations (e.g., due to an anticipated phase change in the application); OS-initiated migrations in response to OS- or hardware-detected phase changes; migration to avoid or ameliorate thermal hot-spots [7, 13]; and migration in response to a fault for unimplemented instruction/feature (such as floating point, vectors, encryption, etc.) on heterogeneous multicores (or on "shared-ISA" multicores [23]).

The improvements we describe can enhance both the effectiveness and the applicability of these techniques, by reducing the overhead of core-switching. As a simple way of evaluating our improvements, this paper specifically examines them in the context of OS-initiated core-switching to an OS-optimized core during expensive system calls. In this scenario, cheaper core-switching both increases the potential gains from using OS-specific cores, and increases the number of system calls for which a core-switch is justified.

1.2 Assumptions

In this paper, we make several assumptions related to core-switching:

- Threads only switch between cores with a single ISA. That is, any core where a thread can run will correctly execute its instructions. (It is possible to trap on certain unimplemented instructions, save the thread state, and resume the thread on a more appropriate core; see Li *et al.* [23] for an example. We leave this case for future work.)
- We only care about the performance consequences of switching a thread between cores on a single die. The code we have developed can successfully switch threads between cores on different sockets, but since inter-socket communication costs are much higher, this is less useful in cases where rapid core-switching is desirable.

2. Related work

In addition to the related work discussed elsewhere in this paper, we summarize research in two areas: thread migration techniques and scheduling for heterogeneous multicores.

2.1 Thread migration techniques

There is a rich history in the literature of systems that support thread migration on conventional multiprocessors [32]. However, the issues and priorities shift significantly on a chip multiprocessor, where extremely low communication costs apply much more pressure on the software overhead of the migration mechanism.

Constantinou *et al.* [9] consider a variety of costs associated with moving threads between cores on a CMP, but focus primarily on moving and warming up state (caches, branch predictors, etc.) and do not address the software costs.

Li *et al.* [22] modify the Linux scheduler to create a custom scheduler, with a load balancer that accounts for the asymmetry of an ASISA system. In particular, they account for the expected costs of moving a particular thread (particularly the cost of moving the cache working set). Their best scheduler migrates threads many orders of magnitude more often than the original Linux scheduler.

Choi *et al.* [8] examine the specific case of migrating branch predictor state when a thread switches cores, but do not address software overhead issues.

Brown and Tullsen [6] take a different approach and propose a "shared-thread" multiprocessor (STMP), where the hardware manages thread movement. Thread state is represented in hardware that is shared among all cores on a chip, so the hardware can move a thread between cores without direct OS involvement. For example, STMP would allow core-switching in a user-mode threads package.

STMP is an intriguing alternative to software-managed core-switching, but we do not yet have the ability to experiment with how this approach interacts with OS code. While the STMP hardware can "reschedule" active threads whose state is represented in hardware, we believe that the OS will have to be involved with scheduling threads in several cases; for example, when threads are blocked on OS-mediated I/O operations, or when there are too many threads for the hardware to represent, or when the scheduling policies must involve OS-managed state. STMP will therefore require fairly significant modifications to the operating system's view of the threads it manages. It is possible that many of the same issues discussed in this paper will need to be addressed for an STMP system.

2.2 Scheduling for heterogeneous multicores

Kumar *et al.* present scheduling policies for ASISA architectures in both the performance [21] and power/performance [20] domains. Those scheduling policies are sampling-based, and run at infrequent intervals to limit migration costs. Their assumption is that migration points are unknown to the software, which is not the case in the architecture that we assume.

Fedorova *et al.* [11], similar to [22], create a custom OS scheduler to account for an asymmetric system. In particular, they seek to achieve a more balanced core assignment to increase fairness and decrease jitter (unpredictable runtimes).

Chakraborty *et al.* [19] also look at migrating OS code to distinct processors. However, their motivation is not heterogeneous cores, but the opportunity to spread computation unlikely to use the same working set (e.g., user and kernel code) onto separate cores with separate caches.

3. Software approaches to core switching

We first describe our basic approach to software core-switching, explaining why all such approaches will involve the kernel's scheduler. We then describe an evolutionary sequence of increasingly more efficient designs. Finally, we describe how we modified the code for various specific system calls to invoke core-switching.

We have implemented all of our changes as modifications to the Linux 2.6.18 kernel. We have tried to keep our changes relatively limited, although our design requires minor changes to various system call implementations (see sec. 3.6) and tightly-integrated changes to scheduling functions.

While 2.6.18 is not the most recent kernel, it uses a scheduler optimized for efficiency [1]. Starting with 2.6.23, Linux uses the "Completely Fair Scheduler" (CFS), whose design the author describes as "quite radical" [26]; we have not yet analyzed this design to how it can be optimized for efficient core-switching, and its increased modularity might add some latency. There is no obvious reason why our approach would not generalize to other operating systems, but our expertise is in Linux.

3.1 Why core-switching is like scheduling

It might seem that moving thread T from core A to core B simply requires transferring the thread's architectural state from A to B. However, this would only work if we are sure that core B is idle before the core-switch, and if we don't want to run another thread on core A while T is bound to (and perhaps blocked at!) core B. Otherwise, we need to ensure that T is currently the most appropriate thread to run on B, and to find another thread (perhaps the idle thread) to run on A. These are both scheduling decisions, requiring access and updates to the kernel's scheduling data structures – this is why the scheduler must be involved in a core-switch.

In all versions of our approach, therefore, kernel code (such as a lengthy system call) that wants to initiate a core-switch calls a **SwitchCores** function that, sooner or later ends up invoking Linux's **schedule** function.

Note that our approach does not replace the operating system's normal scheduling policies. All standard scheduling proceeds as usual (e.g., when a thread's quantum expires), and our code does not core-switch at such points.

3.2 V1: Linux's thread-migration mechanism

Our first approach was to use Linux's *existing* thread-migration mechanism, normally used for relatively long-term load-balancing across cores. To our knowledge, Linux's thread-migration mechanism is the current state-of-the-art for core-switching.

When a task wants to migrate, it (in essence) puts itself on a per-core migration queue, wakes up and switches control to a per-core "migration thread," which does the actual work of moving the thread to the run queue of the target core. If the target core is idle, the migration thread signals that core to invoke the scheduler (see sec. 3.5 for how this is done), which finds the thread on the target run queue and reawakens it. (This mechanism is also invoked when an application uses the **sched_setaffinity** system call in a way that requires it to vacate the current CPU.) Note that this migration approach involves an "extra" context switch, between the initiating thread and the migration thread.

We wrote a version of **SwitchCores**, with less than 30 non-comment source code lines, that invokes this thread-migration mechanism.

3.3 V2: Modified scheduler

We want to remove that extra context switch, and have the initiating thread migrate itself. This means that **SwitchCores** will directly invoke a modified version of Linux's **schedule** function[1]. The changes are not very extensive (43 non-comment source-code lines), but were hard to debug because the scheduler code has many subtle details. (This version, and some of the code described in sec. 3.6, is based on an implementation we first described in [25].)

Since we could not change the arguments to **schedule**, we created a new thread-info field to pass the target CPU (core) ID from **SwitchCores**. (In the Linux scheduler, each core is treated as a distinct CPU.) If this field is set, **schedule** unconditionally deactivates the thread T, then places it on a special per-core queue AQ (for "Alternate Queue"), and then rejoins the original scheduler code where it picks the next thread N to run on the source core. Once **schedule** has context-switched the core to thread N, our modified version checks AQ, finds T there, and (now that T is safely dormant on the source core), inserts it in the run queue for the target core, and signals the target core (see sec. 3.5).

This cross-core signal causes (sooner or later) **schedule** to run on the target core; it finds T on its run-queue, context-switches to T, and the core-switch is complete.

3.4 V3: Scheduler fast-paths

Through analysis of instruction-level traces from simulation runs (see secs. 4 and 5) we discovered that our modified scheduler executes several chunks of slow and unnecessary code. This led us to realize that a fast-path version of **schedule**, tailored to the specific case of core-switching, could run faster.

We actually created three versions: the original modified **schedule**; a **fast_schedule_source** version, called to initiate a core-switch, and a **fast_schedule_target** version, called at the target core in response to the cross-core signal.

Both **fast_schedule_source** and **fast_schedule_target** omit a number of housekeeping functions normally done in **schedule** (e.g., recalculating thread priority, which involves expensive arithmetic, and load-balancing for an about-to-be-idle core). Also, **fast_schedule_source** sets a special per-core hint for the target core, which tells the target's idle loop to invoke **fast_schedule_target** rather than **schedule**. Since this is a hint, we do not have to pay the cost of locking it, although this means that we might occasionally use the slow path (**schedule**) on the target.

Note that standard scheduling events (e.g., on the expiration of a thread's quantum) always use the normal slow-path (**schedule**), and hence all housekeeping functions are executed approximately as often as they would normally be done.

fast_schedule_target cannot omit the code that checks AQ, because although the application thread initiates the core-switch using **fast_schedule_source**, the check of AQ happens after the context-switch, and thus in the idle thread, which called **fast_schedule_target** before it yielded the CPU (and hence its PC is still in **fast_schedule_target**). This is a subtle point; fig. 1 depicts the timeline schematically and shows that **fast_schedule_target** always executes in the context of the idle thread (on one core or another). For the same reason, **fast_schedule_source** can omit the code that checks AQ.

Fig. 2 (left) shows severely abstracted pseudo-code for the scheduler; our basic modifications for core-switching are under-

[1] Apparently, Solaris uses this approach for thread migrations [10, 24].

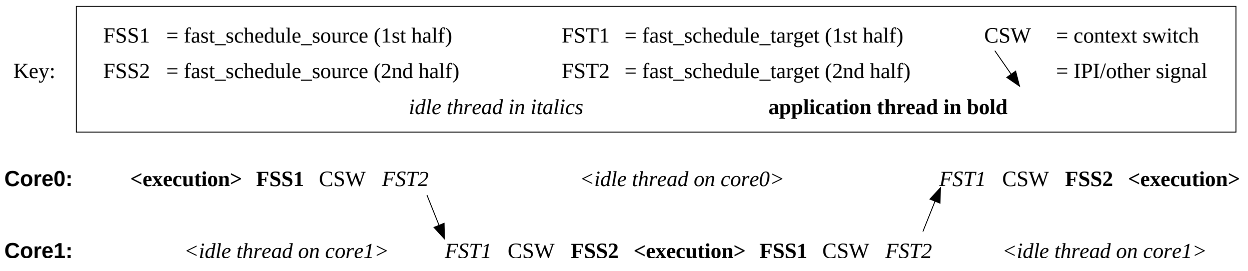
Key:
FSS1 = fast_schedule_source (1st half)
FSS2 = fast_schedule_source (2nd half)
FST1 = fast_schedule_target (1st half)
FST2 = fast_schedule_target (2nd half)
CSW = context switch
= IPI/other signal
idle thread in italics
application thread in bold
Core0:
<execution> FSS1 CSW FST2
<idle thread on core0>
FST1 CSW FSS2 <execution>
Core1:
<idle thread on core1>
FST1 CSW FSS2 <execution> FSS1 CSW FST2
<idle thread on core1>

Figure 1. *Timeline showing 2 core-switches between a pair of cores, using fast-path versions of* ***schedule***

```
void schedule(void) {
sanity checks
prev = current thread;
more sanity checks
compute prev's run-time during quantum
if prev wants to switch cores then
    deactivate prev from runqueue[this_core]
    place prev on AQ[this_core]
else
    handle signal-related state changes for prev
end if
if no other runnable threads for this core then
    try to borrow some threads from other cores
end if
if still nothing to run then
    next = idle
else
    next = highest-priority runnable thread for this core
end if
adjust priority for next
SMT-related optimizations
prefetch next's kernel thread-related state and stack
prepare and do actual context switch
if AQ[this_core] non-empty then
    dequeue thread t from AQ[this_core]
    update load-balancing info for t
    place t on runqueue[t->target_core]
    signal target_core to check its runqueue
end if
}
```

Modifications for core-switching are underlined

```
void fast_schedule_source(void) {
~~sanity checks~~
prev = current thread;
~~more sanity checks~~
~~compute prev's run-time during quantum~~
if prev wants to switch cores then
    deactivate prev from runqueue[this_core]
    place prev on AQ[this_core]
else
    ~~handle signal-related state changes for prev~~
end if
if ~~no other runnable threads for this core~~ then
    ~~try to borrow some threads from other cores~~
end if
if still nothing to run then
    next = idle
else
    next = highest-priority runnable thread for this core
end if
adjust priority for next
~~SMT-related optimizations~~
prefetch next's kernel thread-related state and stack
prepare and do actual context switch
if ~~AQ[this_core] non-empty~~ then
    ~~dequeue thread t from AQ[this_core]~~
    ~~update load-balancing info for t~~
    ~~place t on runqueue[t->target_core]~~
    ~~signal target_core to check its runqueue~~
end if
}
```

Source-core fast-path; deletions are ~~struck out~~

Figure 2. *Abstract pseudo-code for modified versions of* ***schedule()***

lined. Fig. 2 (right) shows how we deleted parts of this code for **fast_schedule_source**.

3.5 V4: Addressing IPI costs

Our modified scheduler needs to wake up the target core if it is currently idle. Linux allows the idle loop either to poll for changes to a "need_resched" flag, or to pause ("quiesce") the core and wait for an inter-processor interrupt (IPI). Architectures that support CPU power-down generally quiesce idle cores. Like the scheduler, the IPI code path includes significant software overhead, which is unnecessary in this instance.

First, the existing code to send an IPI to a specific CPU invokes a function that sends the IPI to all members of a specified set of CPUs. Although the set, in this case, is a singleton, the bit-map manipulations to figure this out take a surprisingly long time. We modified the IPI-sending function to be more efficient. All of our benchmark results for IPI-based core-switching use this improved code.

Second, and more important, using the IPI to invoke **schedule** on the target core results in a long code path for interrupt handling. We thus tested a version ("V4") that forces the use of polling in our simulated system. (Our real x86 system already used polling.)

In sec. 6.1, we discuss a simple hardware change that can yield the time-efficiency of polling without requiring the idle core to stay powered-up, and therefore without the power-increasing drawbacks of V4's naïve polling.

3.6 Modified system calls

Given that our core-switching mechanism is reasonably efficient but not free, when should we switch cores? One approach would be to provide a simple system call (**coreswitch**(destcoreset, flags)) to allow an application to explicitly initiate switching to one of a set of cores, with policies (such as core affinity) controlled by **flags**. This requires application-specific changes, so we have not yet evaluated this approach.

Instead, chose a few long-running and frequently-used system calls, and these calls then invoke core-switching. Fig. 3 shows how

```
sys_read(unsigned int fd, char __user * buf,
                                        size_t count) {
        struct file *file; ssize_t ret = -EBADF;
        int fput_needed; int switched = 0;

/*new*/ if ((count >= RWThresh) && OKtoSwitch(__NR_read))
/*new*/    switched = SwitchToOScore();

        file = fget_light(fd, &fput_needed);
        if (file) {
                loff_t pos = file_pos_read(file);
                ret = vfs_read(file, buf, count, &pos);
                file_pos_write(file, pos);
                fput_light(file, fput_needed);
        }

/*new*/ if (switched)
/*new*/   switched = SwitchToAppCore();

        return ret;
}
```

Figure 3. *Example of system call modified to support core-switching*

(with lines marked `/*new*/`) we added code to the **read** system call, to conditionally switch to an OS-specific core if the buffer size exceeds a threshold. The costly work (mostly in **vfs_read**) then continues on the OS core; when done, if we did switch cores, we switch back.

We modified a number of system calls along these lines: **open, stat, read, write, readv, writev, select, poll, fsync, fdatasync, readfrom, sendto** and **sendfile.** We arbitrarily set the buffer-length threshold to 4096 bytes; we have not yet evaluated whether this is the right setting, or whether it should vary depending on system call or application.

In the future, we would like to experiment with dynamic mechanism for choosing whether to core-switch; for example, based on recent timings for a given system call executed by a given thread. Nellans *et al.* have proposed triggering core-switches using a hardware-based predictor of long privileged-mode instruction sequences, and have shown that such a predictor could be fairly accurate [29].

4. Simulation environment and workloads

Our simulation experiments build upon those we first described in [25]. We used the M5 simulator [4], which supports execution of full systems, including operating system code, and can simulate detailed architectural models cycle-by-cycle.

With M5, we can generate detailed timelines, showing when interesting events such as procedure calls, cache misses, and long-latency instructions occur; these timelines have been valuable in understanding where time is being spent. M5 can also generate detailed traces showing, for example, when specific cores are idle or active.

Since the x86 processor models for M5 are not yet debugged, we used a model based on the Alpha EV6 (21264) as the "complex" core, and an EV4-based (21064) model for the "simple" core. The complex core has 64KB, 2-way set associative, 64B block size L1 caches, with 1-cycle access for the L1 I-cache and 2-cycle access for the L1 D-cache. The simple core has 8KB, direct-mapped, 64B block size, 1-cycle L1 caches. (but see sec. 7, where we evaluate simple cores with larger L1 caches). We simulated a shared L2 cache (3.5 MBytes, 7-way set associative, 4 nsec access), and a main-memory access time of 25 nsec.

We simulated a variety of configurations, with a naming scheme showing the types ("C" for complex, "S" for simple) and speeds (upper-case letters for 3GHz, lower-case for 750 MHz) of each core: **sim_C** with a single (uniprocessor) 3GHz complex core; **sim_CC** with two 3GHz complex cores; **sim_SS** with two 3GHz simple cores; **sim_Cc** with two complex cores; one at 3GHz and one at 750MHz; **sim_CS** with one complex core and one simple core, both at 3GHz; and **sim_Cs** with one 3GHz complex core and one 750MHz simple core. The first letter represents the application core; the second represents the OS core.

For each dual-processor hardware configuration, we ran tests using unmodified Linux, an unmodified Linux that binds Ethernet and disk interrupts to the OS core but does not core-switch, and our five versions of core-switching Linux (**V1 ... V5**), also with interrupts bound to the OS core. For all configurations but the unmodified Linux without interrupt-binding, we pinned the user application process to the application core.

The simulator is fully deterministic, so we ran only one trial per experiment. (One could perhaps introduce some artificial randomness to the simulations to create differences between multiple trials.)

4.1 Modelling core power-up

We expect that a CPU will often have one or more idle cores, and that power could be saved by powering-down these idle cores. (In this paper, we do not consider the specific power savings.) This leads to some issues in modelling the performance aspects of powering up a core.

We assume for these experiments that a core's L1 cache state persists during power-down. Even for our complex core's 64 KB L1 cache, leakage for an inactive cache is only about 0.1 W (from CACTI [33]). Thus, little power is saved by shutting down the cache for short intervals – especially when considering the power cost of refilling the cache. In future work, we will also investigate the case where the L1 cache is flushed when the core is powered down. We also assume that register state is preserved; the "C6" state in the Intel Core Duo [16] suggests this is feasible.

We modified M5 to simulate a configurable delay between the time that a quiesced core receives an interrupt and the time that it executes the first interrupt-handler instruction. See sec. 7.2 for a discussion of this delay and how our results depend on it; simulations presented elsewhere in this paper use a delay of zero.

4.2 Workloads

We ran these OS-intensive benchmarks: **netperf/TCPstream** benchmark, which sends TCP data as fast as possible, and **netperf/TCPmaerts**, which receives data as fast as possible; **Web**, Apache with a workload based on SPECweb; and **DB**, using the **ex_tpcb** "TPC-B-like" example from the Berkeley DB distribution [31].

Table 1. Fraction of CPU time spent in various modes

NIC		Percentage of CPU time spent in mode			
speed	Benchmark	User	Kernel	Interrupt	Idle
1 Gbps	TCPmaerts	14.8%	27.4%	17.5%	40.4%
1	TCPstream	0.1%	38.1%	18.2%	43.7%
1	Web	55.0%	25.8%	10.6%	8.6%
10 Gbps	TCPmaerts	25.0%	46.2%	12.9%	15.9%
10	TCPstream	0.1%	27.0%	25.4%	47.5%
10	Web	54.5%	29.8%	8.3%	7.4%

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

Table 1 shows that these applications spend a lot of their non-idle time in kernel or interrupt modes, although at lower network interface (NIC) speeds the netperf benchmarks often leave the CPU idle.

5. Microbenchmark results

Core-switching is not free; it adds direct overhead due to extra instruction execution, and indirect overhead due to loss of cache affinity. (It can also improve performance if the use of distinct L1 caches on source and target cores reduces the number of conflict and capacity misses.)

We measured the direct overhead using a microbenchmark, which invokes core-switching as rapidly as possible. Since the modified system calls listed in sec. 3.6 were chosen because they spend a lot of time in the kernel, none of these are suitable. Instead, we modified the **gettid** (get thread ID) call, since (unlike **getpid**) it is not cached by the user-mode library, and it executes very few kernel-mode instructions. We then wrote *gettidbench*, which executes **gettid** N times in a tight loop, measures the total elapsed time, and computes the mean time/call. This yields twice the mean time per core-switch, since the system call switches to the OS core and then back to the application core. Note that, on asymmetric hardware, the cost of a single switch may depend on its direction.

5.1 Results on real (x86) hardware

We ran *gettidbench* ($N = 1,000,000$) on a dual-core Xeon model 5160 (3.0GHz, 64-Kbyte L1 caches, 4-MByte L2 cache), with Linux compiled in 32-bit mode.

Table 2(a) shows the results. Even in single-user mode, we saw some variation between trials, so we report the minimum and maximum values for the mean **gettid** delay; we believe the minimum values are more likely to provide a noise-free comparison. With mechanism V3 (scheduler fast-paths), we measured a round-trip (two-core-switch) delay of 2870 nsec, for an excess of 1394 nsec per core switch over the unmodified call.[2] For the entire system call this is a 1.17 speedup over mechanism V2 (modified schedule) and a speedup of 1.43 over the time for mechanism V1 (Linux's thread migration).

5.2 Results on simulated hardware

We ran the benchmark on various configurations of our simulated hardware. In this case, the simulator allowed us to measure the actual (simulated) duration of one call to *gettidbench*, which avoids the noise involved in measuring the average of many trials. Our measured duration came after first warming up the caches with 100 calls. Table 2(b) shows the results.

Note that the delays for the sim_CC configuration are quite similar to those for the real hardware in table 2(a); since both are 3 GHz dual-core CPUs (albeit with different instruction sets), this result helps to validate the simulations.

The lowest delays are for V4 (using polling instead of interrupts); on the sim_CC configuration, this represents an excess of 933 nsec. per core-switch, while on the slowest hardware, sim_Cs, the excess is 3332 nsec. Both of these represent substantial improvements over the existing Linux thread-migration mechanism (V1) and over our unoptimized code (V2).

6. Speeding up software-based core-switching

Here we discuss a number of ways to further improve the speed of software-based core-switching. Some can be implemented entirely in software, others require hardware support.

6.1 Cross-core wakeup from quiesce

As discussed in sec. 3.5, Linux's idle loop either polls on a "need_resched" flag, or waits for a cross-CPU interrupt. The former mechanism is inefficient, especially for cores that can power-down; the latter is slow, because of the interrupt-handler overhead (which we estimate at 160 nsec on our simulated 3GHz Alpha).

[2] Remember that each **gettid** call in this benchmark results in *two* core switches.

Table 3. Microbenchmark results with cross-core wakeup

	gettid delay in nsec.		
Hardware	**Core-switching mechanism**		
config.	**V3**	**V4**	**V5 (wakeup)**
sim_CC	2550	1986	2042
sim_SS	7982	6770	6689
sim_CS	4696	3869	4087
sim_Cs	8140	6781	7024

We therefore added a new cross-core wakeup instruction to our simulated CPU architecture, **wakeup(core)**.[3] This causes the specified core to continue from a quiesce instruction; it is a no-op if that core is not quiesced. Our simulator continues execution on the awakened core after a delay to account for stabilizing the core upon power-up; the delay mechanism is as described in sec. 4.1.

We then modified **fast_schedule_source** to issue a cross-core wakeup instruction as early as possible. This completely hides the target-core stabilization delay, by overlapping it with a considerable amount of instruction execution on the source core, except when the source core is much faster than the target core. When this is sucessful, it avoids the interrupt-handler overhead entirely. (It is possible that we could do the wakeup later in **fast_schedule_source** to save a little power, but it is hard to calculate the optimal point without risking doing this too late.) The target CPU might start running before the "need_resched" flag is set, so we use a separate per-core flag to tell it to temporarily poll rather than quiesce.

The results, shown in fig. 3, reveal that while the version using cross-core wakeups (V5) is not always faster than the always-polling (V4) version, both are consistently faster than the fastest interrupt-based version (V3). While V5 generally follows the same approach as V4, it executes a few extra instructions in the critical path to both issue the cross-core wakeup, and to reset some state flags after the wakeup takes effect. Generally, given these similar delays, cross-core wakeups will be more energy-efficient than polling if idle cores can be powered down.

6.2 Cross-core task-state prefetch

As shown in fig. 2, just before context-switching the **schedule** code attempts to prefetch the new thread's kernel thread state and stack. Not all architectures support such prefetching (apparently, only IA64 supports the stack-prefetch), and as far as we can tell, the code only attempts to prefetch the first cache line of the thread state.

It might be possible to further speed up a core-switch if the source core could ask the target core to prefetch thread state into the target's L1 cache, as soon as **fast_schedule_source** knows that this is necessary. This could possibly be combined with the cross-core wakeup, and we would almost certainly want to prefetch more than one cache line of this state.

6.3 Conditional use of power-down

Powering down a core could hurt performance (even with cross-core wakeup) if this requires flushing the core's L1 caches. Therefore, the kernel should only power-down a core if it is likely to be idle for a while. The problem is how to predict whether it is advantageous to power-down or not. This problem is analogous to the decision to use spin-wait or blocking locks, where Karlin *et al.* [18] showed that the use of *competitive algorithms* can yield good results. These algorithms dynamically decide between spinning and blocking based on recent history. For core-switching systems, recent history could also be a good guide to the duration of future

[3] This is analogous to the Intel monitor/mwait mechanism. We do not claim that our approach is better than Intel's, but it is simpler to simulate. Our goal is to show how core-switching software can exploit this kind of feature.

Table 2. Microbenchmark results

(a) dual-core x86 hardware

Over 10 trials	Mean gettid delay in nsec. Core-switching mechanism None	V1	V2	V3
min.	83	4094	3355	2870
max.	87	4320	3413	2886

Per-call delay computed from 1,000,000 iterations of the gettid call per trial.

(b) simulated hardware

Hardware config.	gettid delay in nsec. Core-switching mechanism None	V1	V2	V3	V4
sim_CC	120	4669	3247	2550	1986
sim_SS	130	13229	10248	7982	6770
sim_CS	121	8651	6343	4696	3869
sim_Cs	118	16735	11148	8140	6781

idle-core intervals. Finding the right mechanism to decide dynamically whether to power down is the subject of future research.

6.4 Summary of core-switching versions

Table 4. Summary of core-switching versions

Version number	Mechanism(s) used
V1	Linux's existing thread-migration mechanism
V2	Direct invocation of modified scheduler
V3	Scheduler fast-paths for source and target
V4	Idle loop uses polling instead of interrupts
V5	Cross-core wakeup from quiesce

Table 4 summarizes the mechanisms used in the five different core-switching versions we implemented.

7. Effects of architectural parameters

We looked at the effect of two architectural parameters on the performance of our simulated microbenchmarks: L1 cache size, and core-wakeup delay.

7.1 L1 cache sizes

Table 5. Effect of L1 cache size on microbenchmark results

Hardware config.	gettid delay in nsec. 8 KB L1 caches	16 KB L1 caches	16 KB 2-way L1 caches
sim_SS	6689	5692	3787
sim_CS	4087	3665	2706
sim_Cs	7024	6515	5515

Results based on V5 core-switching mechanism

The cost of migration will be sensitive to the sizes of the caches (both instruction and data). The L1 caches (8KB, direct-mapped, 64B block size, 1-cycle) for our simple cores are relatively small, so we also simulated two versions of larger caches. Both are 16KB total size, 64B block size. One is direct-mapped, the other is two-way set associative. Table 5 shows that, for the V5 mechanism, increasing the simple-core L1 cache size to 16KB does indeed improve performance, by 7% (for sim_Cs, with one slow simple core) to 15% (for sim_SS, with one fast simple core). (There is no line in this table for sim_CC, since that configuration has only complex cores, which we always model with 64KB, 2-way L1 caches.)

Adding associativity further improves performance, by 15% for sim_Cs to 33% for sim_SS. The large impact of associtivity implies that we are seeing a lot of conflict misses.

Based on our examination of detailed miss-rate statistics, we speculate that the 8KB I-cache creates lots of capacity misses, which are mostly eliminated by the 16KB direct-mapped cache. However, the D-cache miss rate benefits both from the larger size and again from the associativity, implying that it suffers from the conflict misses.

7.2 Core-wakeup delay

When a powered-down core is powered up, some time passes until the voltages have stabilized enough for the core to safely execute instructions. This delay is imposed by the RC time constant defined by the capacitance of the core and the resistance in the power wiring. We used a delay model provide by Matteo Monchiero [27]; with parameters chosen for a 65-nm process and a maximum core current of 10A, the model predicts a power-on time of about 16.3 nsec, or about 50 cycles at 3 GHz and 12.5 cycles at 750 MHz. For comparison, James *et al.* report that "a single POWER6 core is capable of causing a 13W power step within about 20 clock cycles" [17].

We ran M5 simulations in which the wakeup delay was very conservatively set to 1000 cycles for all cores (at both 3 GHz and 750 MHz). Table 6 shows the results. Increasing the delay has no effect on the non-switching and V4 (always-polling) versions, since neither of these ever quiesces (powers down) a core. Similarly, it adds roughly the expected delay for the V1, V2, and V3 configurations (333 nsec to wake up the 3GHz cores, and 1333 nsec to wake up the 750 MHz core). When using the cross-core wakeup (V5), there is essentially no effect from the added wakeup delay, since the wakeup is generated much earlier than necessary to cover 1000 cycles (and, therefore, to cover a shorter, more realistic delay).

In the one case of the sim_Cs configuration, where one core is much faster than the other, when switching from the fast core to the slow core, the cross-core wakeup does not happen soon enough to finish the 1000-cycle delay before the IPI is generated. We also simulated a 250-cycle delay, and found that in this case, the core does wake up before the IPI is sent, but just slightly too late to set the "I am polling" flag before the source core decides to send the IPI.

8. Macrobenchmark results

We ran a variety of "macrobenchmarks" on both the real dual-core x86 and on a variety of simulated configurations. These are relatively simple throughput-oriented benchmarks, which represent realistic execution scenarios and are designed to stress the code that core-switches during lengthy system calls. In the simulations, this means running some OS code on the simpler (or slower) core; on the real hardware, both cores are equally fast.

These macrobenchmarks do not, unfortunately, prove that "real applications" in general can profit from frequent low-latency core switching. To do so would require simulations with realistic workloads, which would be much lengthier than we have been able to support to date.

Remember also that the core-switching configurations are not meant to give better throughput than the non-switching systems, but rather to enable more frequent power-down of complex cores. Therefore, in our macrobenchmark results, we are not looking for throughput improvements from our various implementations of core-switching; we only care that they do not significantly reduce performance.

Table 6. Effect of power-up delay on performance

HW config.	Wakeup delay (cycles)	gettid delay in nsec. Core-switching mechanism None	V1	V2	V3	V4	V5
sim_CC	0	120	4669	3247	2550	1986	2042
sim_CC	1000	120	5216	3914	3224	1986	2059
sim_SS	0	130	13229	10248	7982	6770	6689
sim_SS	1000	130	13977	10916	8630	6770	6690
sim_CS	0	121	8651	6343	4696	3869	4087
sim_CS	1000	121	9376	6847	5393	3869	4081
sim_Cs	0	118	16735	11148	8140	6781	7024
sim_Cs	250	118	16891	11480	8555	6781	7930
sim_Cs	1000	118	16514	12835	9822	6781	9503

Table 9. Core-switch counts for 1 Web trial, dual-core X86

Benchmark	Core-switches by system call read	write	open	fsync	poll	sendfile
Web	2131	7	1829	6	3649	3640

8.1 Simulation pitfalls

Simulations of TCP-based benchmarks can be problematic, as described by Hsu *et al.* [15]. TCP performance depends on, among other things, the apparent round-trip time (RTT); changes in the RTT during a connection can cause packet losses or spurious retransmissions. Unfortunately, some of the techniques we use to make simulations more efficient cause changes in apparent RTT. We boot the system and start the benchmark in a fast simulation mode, then checkpoint and switch to a detailed simulation, possibly using slower CPU cores. This sudden change in CPU speed can lead to increased software delays in packet processing, causing a sudden increase in the apparent RTT.

We took some steps to avoid this problem, such as slowing down the simulated CPU clock speeds during the pre-checkpoint phase, and increasing the simulated LAN latency; these steps mean that the relative effects on RTT of core speed are reduced. However, increasing the total RTT too much means that TCP connections do not ramp up fast enough, leaving the CPUs idle.

8.2 Web benchmark

We simulated the **Web** benchmark, which is Apache with a workload loosely based on SPECweb[4], for a simulated time of 133 msec., after a warmup period of 333 msec. using M5's simpler CPU model. Table 7 shows the results. The NIC speed has no significant effect, because this benchmark nearly saturates the CPUs, as shown in table 1. Note that all of the "bound" configurations, with the exception of sim_SS, achieve essentially the same throughput, because (except for sim_SS) all of these are executing non-interrupt code on a full-speed complex core. (One trial failed due to a simulator bug.)

For this benchmark, core-switching imposes a fairly significant cost depending on the configuration of the OS core. (A fast The results show that the **V2**–**V5** kernels generally outperform the **V1** kernel, but it is not clear whether the differences between the **V2**–**V5** kernels themselves are consistent.

[4] Very loosely, it turns out. We discovered that all responses are "404 Not found" due to a buggy mod_specweb99.so. For complex reasons, we cannot fix this bug. It has the effect of making the benchmark more latency-sensitive, since the responses are all short, and it also makes comparisons between relatively brief simulation trials simpler, because all responses are about the same length. However, in order to trigger any core-switching at all on server response transmissions, we therefore set the core-switching threshold for reads and writes to 256B, rather than 4KB. This setting is possibly too aggressive for efficient operation, but it ensures that this benchmark does reflect core-switching costs.

Table 10. Simulated throughput for **ex_tpcb**

HW config.	Kernel configuration Unmod	Bound	V1	V2	V3	V4	V5
sim_C	11411	NA	NA	NA	NA	NA	NA
sim_CC	10467	12006	7251	3291	3002	11416	11631
sim_SS	4562	4789	4307	4355	4402	4413	4419
sim_Cc	3819	10541	7276	7419	7615	7627	7746
sim_CS	4563	10320	8556	8680	8872	8857	8952
sim_CS*	5510	11644	9092	9233	9422	9439	9619
sim_Cs	2336	9007	5930	5900	6054	6063	6298
sim_Cs*	2608	9442	6240	6098	6240	6258	6306

Values are transactions/sec. rates (for 100 transactions)
*: this trial used 16KB L1 caches

We also ran this benchmark on the dual-core Xeon hardware, using trials of 15 seconds. Again, the throughput was identical, saturating the 1 Gbit/sec NIC, independent of kernel configuration.

Table 9 shows that several different system calls accounted for the majority of the core-switches during this benchmark. (**poll** is normally used to wait for available input; **sendfile** is used to transmit data directly from a file to the network, without copying it into and out of user space.) We note, however, that in profiles made on the simulated system, only the **writev** system call consumes significant CPU time, possibly implying that Apache on the simulated system is not using **sendfile**.

We also simulated quad-core configurations (using the notation from sec. 4, sim_CCSS, sim_CCCC, and sim_CCss), where Apache is bound to the two complex cores, and system calls and interrupts are bound to the two other cores. The results in Table 8 show that in these tests, the fastest core-switching kernel is **V4**, as we would expect, but there is little difference between the **V2**–**V5** kernels. (One trial failed due to an M5 bug, and another died with a kernel crash, apparently because of a synchronization bug in our core-switching code which allowed a timer interrupt during a critical section.) As explained in sec. 9.2, we have not yet really solved the scheduling problem for core-switching with multiple choices of targets, but profiles show a rough balance of effort between the two OS cores.

8.3 Database benchmark

Our **DB** benchmark uses the **ex_tpcb** "TPC-B-like" example from the Berkeley DB distribution [31]. (One should not mistake this for a full-scale database.) Normally, **ex_tpcb**'s throughput is dominated by disk I/O, which makes it hard to evaluate the cost of computation. We eliminated most disk I/O delays by using a RAM disk on the real hardware, and by setting the access time to zero in M5's disk simulator.

In these tests, the configuration apparently handled all interrupts on the OS, so while the other configurations differ from **unmod** in that they pin user-mode code to the application core, they do not actually change where the interrupts happen.

Table 7. Simulated Web results on dual-core CPUs

Hardware config.	NIC speed = 1 GBit/sec							NIC speed = 10 GBit/sec						
	Kernel configuration													
	Unmod.	Bound	V1	V2	V3	V4	V5	Unmod.	Bound	V1	V2	V3	V4	V5
sim_C	1231	NA	NA	NA	NA	NA	NA	1229	NA	NA	NA	NA	NA	NA
sim_CC	2087	1354	706	486	1300	1329	1301	2326	1339	1046	1261	1296	1319	1117
sim_SS	1079	615	490	572	578	585	583	1062	617	476	567	585	587	594
sim_Cc	1560	1393	724	780	852	870	(M5 bug)	1629	1391	579	155	701	696	938
sim_CS	1706	1340	847	1163	1184	1176	1161	1759	1339	872	1162	1171	1178	1169
sim_Cs	1417	1373	482	696	690	704	688	1481	1379	481	695	712	700	703
sim_CS (16KB L1)	1772	1344	953	1192	1211	1204	1196	1777	1330	964	1191	1199	1204	1196
sim_Cs (16KB L1)	1464	1382	574	734	764	762	754	1505	1375	513	764	791	774	753

Values are KB transferred during 0.133 seconds.

Table 8. Simulated Web results on quad-core CPUs

Hardware config.	NIC speed = 1 GBit/sec							NIC speed = 10 GBit/sec						
	Kernel configuration													
	Unmod.	Bound	V1	V2	V3	V4	V5	Unmod.	Bound	V1	V2	V3	V4	V5
sim_CCSS	2897	2739	685	1187	1222	1239	1227	2937	2795	683	1192	1227	1238	1233
sim_CCCC	3425	3431	1318	1278	1301	1322	(M5 bug)	3335	3390	1326	(crashed)	1318	1330	1293
sim_CCss	2175	1704	348	1100	1133	1142	1140	2139	1758	350	1104	1136	1149	1134

Values are KB transferred during 0.133 seconds.

Table 10 shows the results, for trials of 100 transactions, on various simulated hardware configurations. (For this benchmark, we found it unnecessary to use a warmup period, and we core-switched only on **fdatasync** and not on other system calls.) Even though the application is single-threaded, and actually shows a slight slowdown when going from one to two cores under the vanilla kernel, we do see some speedup when using either the **bound** configuration or (in some cases) core-switching. (We are not sure why the V2 and V3 configurations for sim_CC perform so poorly.)

Table 10 also shows that, in general, the unmodified kernel is the worst performer. This is mostly an artifact of the single-threaded application, which on the unmodified kernel tends to run on core 0; in our configurations, the slower core is numbered 0. When we bind the user-mode code to the faster core, this leads to an artificial speedup (except for sim_SS and sim_CC, which have no "faster" core). In the cases of sim_SS and sim_CC, pinning the user code to core 1 has the effect of separating its execution from the interrupts on core 0, and the modest speedup of **bound** over **unmod** may be the result of more parallelism and/or fewer cache conflicts.

Somewhat surprisingly, the V5 kernel usually performs better than the V4 kernel; perhaps the idle-loop polling used in V4 causes interference with the non-idle core.

Table 11 shows the results, for trials of 20,000 transactions, on the dual-core Xeon hardware. As far as we can tell, there is no meaningful impact of the core-switching code on this benchmark, even though it spends ca. 33% of its time in the operating system.

Table 12 shows that essentially all of the core-switches happen during the **fdatasync** system call, which flushes a file's kernel buffers to the disk (and hence makes a transaction durable).

We also obtained procedure profiles for this benchmark from the M5 simulator. For the uniprocessor (sim_C) trial, the system spent 56% of the CPU in user mode, 24% in system calls (including 13% in **fdatasync**), and 8% in interrupt handlers. For the unmodified kernel on the sim_CC dual processor, the primary CPU spent 53% of its time in user mode, 22% in system calls (12% in **fdatasync**), and 7% in interrupt code; the secondary CPU was mostly idle.

For comparison, we looked at the sim_CS CPU (with 8KB L1 caches). For the V5 kernel, the application core spent 42% of it time in user mode, 43% idle, 9% in system calls, and negligible time in interrupts or **fdatasync**. The OS core spent 55% of its time idle, 15% in interrupt code, and 24% in system calls – almost entirely in **fdatasync**. Thus, even though the simple OS core spends more CPU time executing **fdatasync** than a complex core does, there is enough spare OS core time to maintain throughput, while allowing the high-power complex core to be quiesced (in low-power mode) for almost half of the time.

Table 11. Throughput for **ex_tpcb** on dual-core X86

Core-switching mechanism	$N = 100$ **trials** 20,000 transactions/trial		
	Mean TPS	Std. dev.	Max. TPS
None	21532	71	21692
V1	21462	78	21623
V2	21467	85	21583
V3	21491	62	21619
V3, work-conserving	21446	75	21576

Table 12. Core-switch counts for 1 ex_tpcb trial, dual-core X86

Benchmark	Core-switches by system call			
	read	write	open	fdatasync
ex_tpcb	81	23	196	200029

8.4 Network streaming benchmarks

We simulated the **netperf/TCPstream** benchmark, which sends TCP data as fast as possible, and **netperf/TCPmaerts**, which receives data as fast as possible. These ran for a simulated time of 167 msec., after a 33msec warmup period. Tables 13 and 14, respectively, show the results. and our **V5** core-switching kernel (also with bound interrupts). Except for the **Unmod** trials, the application itself was bound to core 1.

In tests with a 1 Gbit/sec, **TCPstream** on the unmodified kernel got essentially full wire bandwidth, except on the sim_SS configuration, which with two simple cores seems underpowered. Core-switching clearly causes a slowdown on CPUs with a slow or simple OS core. There is some variation based on the core-switching version.

There is no clear pattern as to which core-switching kernel performs the best. The benchmarks are configured to send and receive data in buffers of 64KB per system call, which makes core-switching relatively infrequent. Perhaps the variation between trials is due to the effects described in section 8.1. The use of core-switching, however, does move significant CPU time from the fast core to the slow core, which reduces throughput. Also, since almost

Table 13. Simulated Netperf results: TCPstream

Hardware config.	NIC speed = 1 GBit/sec							NIC speed = 10 GBit/sec						
	Kernel configuration													
	Unmod.	**Bound**	**V1**	**V2**	**V3**	**V4**	**V5**	**Unmod.**	**Bound**	**V1**	**V2**	**V3**	**V4**	**V5**
sim_C	21185	NA	NA	NA	NA	NA	NA	47639	NA	NA	NA	NA	NA	NA
sim_CC	21185	21185	21185	21185	21185	21185	21185	46623	65049	47158	47823	48378	48536	48031
sim_SS	13920	19196	14354	14377	13965	14003	13954	13861	19162	14334	14404	13909	13920	13958
sim_Cc	21185	21185	13546	13683	13698	13769	13734	46180	41563	13468	13680	13728	13778	13746
sim_CS	21185	21185	14413	14471	13952	13948	14037	46185	60718	14421	14484	13951	14002	14014
sim_Cs	21185	20939	7139	7128	6990	6973	6990	46120	23103	7105	7139	6987	6987	7021
sim_CS (16KB L1)	21185	21185	17287	17734	17871	17828	18101	46172	62680	17357	17700	17766	17827	18046
sim_Cs (16KB L1)	21185	21185	7845	7970	7965	8030	8036	46146	26547	7836	7953	7965	8002	8039

Values are KB transferred during 0.167 seconds.

Table 14. Simulated Netperf results: TCPmaerts

Hardware config.	NIC speed = 1 GBit/sec							NIC speed = 10 GBit/sec						
	Kernel configuration													
	Unmod.	**Bound**	**V1**	**V2**	**V3**	**V4**	**V5**	**Unmod.**	**Bound**	**V1**	**V2**	**V3**	**V4**	**V5**
sim_C	17793	NA	NA	NA	NA	NA	NA	20417	NA	NA	NA	NA	NA	NA
sim_CC	17668	16549	16302	16424	16425	16425	16425	20266	18966	18688	18826	18827	18827	18827
sim_SS	17268	16545	16018	16144	16144	16148	16147	18655	19018	17164	17840	17852	17913	17913
sim_CS	17668	16554	16020	16140	16145	16144	16166	20266	18966	17751	17960	17960	18023	18023
sim_Cs	21185	13546	11382	11458	11494	11499	11494	24155	13847	11631	11829	11902	11902	11902

Values are KB transferred during 0.167 seconds.

no time is spent in user mode, the application core is almost entirely idle, and need not be drawing power.

At 10 Gbit/sec, we could not saturate the network; the CPU was the bottleneck. Interestingly, on the sim_CC configuration, the **bound** configuration yielded 40% more throughput than the unmodified configuration. A profile of the unmodified configuration shows that the scheduler has put both the user code and interrupt handling on the same core, leaving the other core fully idle except for clock ticks. The **bound** configuration spread the load somewhat more evenly over both cores. Again, core-switching does cause a slowdown for CPUs with slower OS cores, and there is no clear winner among the core-switching kernels. Core-switching and interrupt-binding forces most of the load onto the OS core, again leaving the application core mostly idle.

With **TCPmaerts**, the results seem inconsistent, possibly because of the effects described in section 8.1; profiles show even the uniprocessor CPU mostly idle. Table 14 therefore omits some rows, to save space.

Table 15. Core-switch counts for 1 netperf trial, dual-core X86

Benchmark	Core-switches by system call				
	read	**write**	**open**	**socketcall**	**poll**
netperf/tcpstream	273	0	120	431216	10
netperf/tcpmaerts	410	1	460	431240	10

On the real dual-core Xeon system, with a 1 Gbit/sec NIC, multiple trials of both streaming benchmarks always transferred between 941.2 and 941.45 Mbit/sec regardless of the software configuration (core-switching or unmodified Linux), implying that the system was network-limited. (We have no 10 Gbit/sec NIC for this system.) We did measure the number of times, during one 60-second trial of each benchmark, various system calls performed core-switches. Table 15 shows that almost all of these core switches were in the **socketcall** system call. (Linux on x86 differs from Linux on Alpha in that its C library funnels the socket API through this one system call.)

9. Future work

Here we discuss several aspects of our ongoing work.

9.1 Energy measurements

Our primary motivation (based on Kumar *et al.* [20]) for rapid core switching is to enable the potential energy efficiencies of asymmetric multicores. In this paper, we have not reported energy savings, because we do not yet have accurate energy models for asymmetric hardware. We are in the process of porting the Wattch power-analysis framework [5] to M5, which should allow reasonably accurate modelling of dynamic power consumption.

9.2 Policies for core switching

Core-switching introduces policy questions: (1) which core to choose, and (2) whether to switch to a core that is busy. Here we describe some preliminary work that addresses those issues; we do not yet have the full experimental results that would show which policies are the most efficient.

On a two-core machine, the first question is easy; the second reduces to whether the policy is *work-conserving*. A work-conserving policy would not switch a thread to a busy target core, causing the thread to block, if the source core would become idle. It is not obvious whether a work-conserving policy would decrease overall energy efficiency (by doing work on a more-complex core than necessary) or increase it (by finishing the task sooner, and thus spending less energy on always-on system components).

On a machine with more than one core of each type, the which-core-to-choose question arises. Policy options include static binding (e.g., threads on application core X_a always switch to OS core X_o); round-robin (for load balancing), or affinity-favoring. If the policy both favors affinity and is work-conserving, there is also a choice as to how to break ties (i.e., should we switch to the same core as before even if that core is busy?)

We implemented a many-core version of **SwitchCores** that invokes a **SelectOSCore** function controlled by three policy settings: `WC` (for work-conserving), `AF` (for affinity instead of round-robin), and `PA` (for preferring affinity over work-conserving). **SwitchCores** also invokes a corresponding **SelectAppCore** function, analogous to **SelectOSCore** except that it never chooses to leave the thread running on an OS core.

The resulting code has a complex decision tree. The most expensive aspect is the need to search for idle cores when doing round-robin scheduling, but, we have not yet seen any significant difference in overhead versus our simpler scheduling policy.

10. Summary

Core-switching costs will become increasingly important with the use of asymmetric multicores. In this paper, we showed how to significantly speed up core-switching using both software techniques and small architectural changes. We evaluated these changes using both microbenchmarks and macrobenchmarks on both real and simulated hardware. Core-switching to slower OS cores on frequent, expensive system calls sometimes reduces performance – and sometimes improves performance – but it also provides opportunities to power-down complex application cores. We also discussed several scheduling-policy issues that arise from core-switching.

Acknowledgments

We could not have done this work without significant help from Rakesh Kumar, Partha Ranganathan, Vanish Talwar, and the members of the M5 community. We are grateful for the helpful suggestions we received from Alexandra Fedorova, David Nellans, and various anonymous reviewers.

References

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

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proc. ISCA*, pages 506–517, 2005.

[3] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. *J. Instruction-Level Parallelism*, pages 1–26, June 2008.

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proc. ISCA*, pages 83–94, Jun. 2000.

[6] J. A. Brown and D. M. Tullsen. The Shared-Thread Multiprocessor. In *Proc. ICS*, pages 73–82, 2008.

[7] P. Chaparro, J. Gonzalez, and A. Gonzalez. Thermal-Aware Clustered Microarchitectures. In *Proc. ICCD*, pages 48–53, 2004.

[8] B. Choi, L. Porter, and D. Tullsen. Accurate Branch Prediction for Short Threads. In *Proc. ASPLOS*, Apr. 2008.

[9] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance Implications of single thread migration on a chip multi-core. In *Workshop on Design, Arch., and Simulation of Chip Multiprocessors*, Nov. 2005.

[10] A. Fedorova. Personal communication, 2009.

[11] A. Fedorova, D. Vengerov, and D. Doucette. Operating System Scheduling On Heterogeneous Core Systems. In *Proc. Workshop on Op. Sys. Support for Heterogeneous Multicore Architectures*, 2007.

[12] R. E. Grant and A. Afsahi. Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications. In *Proc. Intl. Parallel and Distributed Processing Symp.*, Apr. 2006.

[13] S. Heo, K. Barr, and K. Asanovic. Reducing Power Density through Activity Migration. In *Proc. Intl. Symp. on Low Power Electronic Design*, Aug. 2003.

[14] M. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, Jul. 2008.

[15] L. R. Hsu, A. G. Saidi, N. L. Binkert, and S. K. Reinhardt. Sampling and Stability in TCP/IP cworkloads. In *Proc. Workshop on Modeling, Benchmarking, and Simulation*, Jun. 2005.

[16] Intel Corp. Intel Core 2 Duo Processors and Intel Core 2 Extreme Processors on 45-nm Process: Datasheet. Document Number 320120, Jul 2008.

[17] N. James, P. Restle, J. Friedrich, B. Huott, and B. McCredie. Comparison of Split-Versus Connected-Core Supplies in the POWER6 Microprocessor. In *Proc. ISSCC*, pages 298–604, Feb. 2007.

[18] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. SOSP*, pages 41–55, 1991.

[19] 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.

[20] 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.

[21] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proc. ISCA-31*, June 2004.

[22] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proc. Supercomputing*, pages 1–11, 2007.

[23] T. Li, P. Brett, B. Hohlt, R. Knauerhase, S. D. McElderry, and S. Hahn. Operating System Support for Shared-ISA Asymmetric Multi-core Architectures. In *Workshop on the Interaction between Op. Sys. and Computer Arch.*, June 2008.

[24] R. McDougall and J. Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Prentice Hall, 2nd edition, 2007.

[25] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*, 8(3):26–41, May/June 2008.

[26] I. Molnar. Modular Scheduler Core and Completely Fair Scheduler. `http://kerneltrap.org/node/8059`, Apr. 2007.

[27] M. Monchiero. Personal communication, 2008.

[28] 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.

[29] D. Nellans, R. Balasubramonian, and E. Brunvand. Interference Aware Cache Designs for Operating System Execution. Tech. Rep. UUCS-09-002, University of Utah, Feb. 2009.

[30] D. Nellans, R. Balasubramonian, and E. Brunvand. OS Execution on Multi-Cores: Is Out-Sourcing Worthwhile? *Op. Sys. Review*, 43, Apr. 2009.

[31] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. FREENIX Track, USENIX Annual Tech. Conf.*, pages 183–191, 1999.

[32] J. M. Smith. A survey of process migration mechanisms. *ACM Operating System Review*, July 1998.

[33] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, HP Laboratories Palo Alto, 2006.

[34] B. Wun and P. Crowley. Network I/O Acceleration in Heterogeneous Multicore Processors. In *Proc. 14th IEEE Symp. on High-Performance Interconnects*, pages 9–14, Palo Alto, CA, Aug. 2006.