Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures

Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn

{tong.n.li,paul.brett,rob.knauerhase,david.a.koufaty,dheeraj.reddy,scott.hahn}@intel.com
Intel Corporation

Abstract

A heterogeneous processor consists of cores that are asymmetric in performance and functionality. Such a design provides a cost-effective solution for processor manufacturers to continuously improve both single-thread performance and multi-thread throughput. This design, however, faces significant challenges in the operating system, which traditionally assumes only homogeneous hardware. This paper presents a comprehensive study of OS support for heterogeneous architectures in which cores have asymmetric performance and overlapping, but non-identical instruction sets. Our algorithms allow applications to transparently execute and fairly share different types of cores. We have implemented these algorithms in the Linux 2.6.24 kernel and evaluated them on an actual heterogeneous platform. Evaluation results demonstrate that our designs efficiently manage heterogeneous hardware and enable significant performance improvements for a range of applications.

1 Introduction

Advances in silicon technology have enabled processor manufacturers to integrate more and more cores on a chip. Most multi-core processors consist of identical cores, where each core implements sophisticated microarchitecture techniques, such as superscalar and out-of-order execution, to achieve high single-thread performance. This approach can incur high energy costs as the number of cores continues to grow. Alternatively, a processor can contain many simple, low-power cores, possibly with in-order execution. This approach, however, sacrifices single-thread performance and benefits only applications with thread-level parallelism.

A heterogeneous processor integrates a mix of "big" and "small" cores, and thus can potentially achieve the benefits of both. Several usages motivate this design:

- *Parallel processing*: with a few big and many small cores, the processor can deliver higher performance at possibly the same or lower power than an iso-area homogeneous design. To illustrate the performance benefits, we emulated a heterogeneous processor with one big core and four small cores using a multiprocessor system (details in Section 5), where the big core runs at 2.66 GHz with a 4 MB L2 cache and each small core has a lower frequency, fewer instruction execution units, and a 2 MB L2. Assuming one big core is of equal area to four small cores, we compared this design to a homogeneous one with two big cores only. Figure 1 shows our results. For the heterogeneous design, we also varied the small core frequency, resulting in four configurations as shown on the x-axis. We see that all of our benchmarks obtain higher performance from at least one heterogeneous configuration.
- *Power savings*: the processor uses small cores to save power. For example, it can operate in two modes: a high-power mode in which all cores are available and a low-power mode in which applications only run on the small cores to save power at the cost of performance.

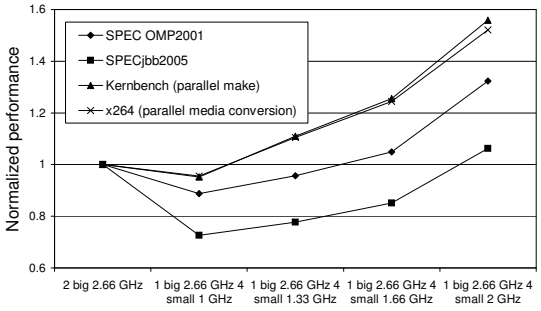
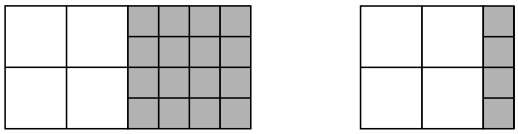


Figure 1: Performance comparisons for two big cores only vs. one big core, and four small cores at different frequencies.



(a) Server product. (b) Client product.

Figure 2: Example server and client processors. Big (small) squares represent big (small) cores with a 1:4 area ratio.

- *Accelerator*: unlike the previous models, where the big cores have higher performance and even more features, in this model, the small cores implement special instructions, such as vector processing, which are unavailable on the big cores. Thus, applications can use the small cores as accelerators for these operations.

These usages are not disjoint. For example, in the parallel processing model, the small cores can also implement unique vector instructions and act as accelerators. There are typically more small cores than big cores in this model, but not necessarily in the others. We expect future designs to have a variety of big-small core ratios, possibly targeting different market segments, as exemplified in Figure 2.

Despite their benefits, heterogeneous architectures pose significant challenges to OS design, which has traditionally assumed homogeneous hardware. This paper studies OS support for heterogeneous architectures in which cores have asymmetric performance and overlapping, but non-identical instruction sets. We propose three algorithms that enable applications to transparently execute and fairly share the heterogeneous cores. The first algorithm, fault-and-migrate, allows applications to run transparently without distinguishing the instruction sets on different cores. The second algorithm, faster-first scheduling, improves performance by scheduling threads to faster cores first. Finally, we extend an SMP algorithm, distributed weighted round-robin (DWRR) [19] to enable fair scheduling for performance-asymmetric cores. Different from previous work, we have implemented our designs in an actual OS and evaluated them on a heterogeneous platform with performance and instruction-based asymmetry. Our experience demonstrates that, with moderate changes, an existing OS can be extended to effectively support heterogeneous hardware.

The remainder of this paper is organized as follows. In Section 2, we discuss our architecture model and OS challenges. Section 3 describes our OS algorithms to support cores with overlapping, but non-identical instruction sets. Section 4 introduces two algorithms to handle performance asymmetry. Section 5 discusses our hardware prototype and Linux implementation. We present evaluation results in Section 6 and discuss related work in Section 7. Finally, we conclude in Section 8.

2 Heterogeneous Architectures

2.1 Design Space

We classify heterogeneous architectures into two types: *performance asymmetry* and *functional asymmetry*. The former refers to architectures where cores differ in performance (and power) due to different clock speeds, cache sizes, microarchitectures, and so forth. Applications run correctly on any core, but can have different performance.

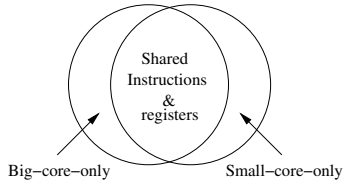


Figure 3: Illustration for instruction-based asymmetry.

Functional asymmetry refers to cores with non-identical ISAs. For example, some cores may be general-purpose while others are fixed-function. General-purpose cores can also have different functionality due to ISA differences. For example, to reduce area, a processor may support vector instructions only on a subset of cores. We use the term ISA to refer to the portion of a core that is visible to software, including instructions, architectural registers, data types, addressing modes, memory architecture, exception and interrupt handling, and external I/O [25]. Without adequate support, programs compiled for one ISA can fail on cores with a different ISA, even when the difference is small.

There are multiple dimensions of functional asymmetry, one for each aspect of the ISA. In the extreme case, a processor contains cores with disjoint ISAs, such as Intel® IXP processors [29] and some implementations of integrated CPU and GPU cores. Alternatively, cores can have overlapping ISAs. The Cell* processor is an example where cores differ in most aspects of the ISA, but share the same data types and virtual memory architecture [12]. Due to disjoint instruction sets, Cell* requires significant programming efforts for the OS, compilers, libraries, and so forth.

We focus on an architecture model that exhibits performance asymmetry and a form of functional asymmetry, *overlapping-ISA asymmetry*, where cores are identical in every aspect of the ISA except a set of instructions and architectural registers, as Figure 3 illustrates. In our model, cores share a large set of common instructions and registers with identical encoding and semantics. For example, if two cores support the same opcode, it must behave identically on the two cores. Additionally, each core can implement a small set of instructions unique to its own core type. Like current CMPs, we assume that all cores share a physical address space with coherent caches. For simplicity, our discussion assumes only two core types: big and small; our algorithms, however, are applicable to more types of cores.

We believe that this model is a likely choice for future designs. First, all cores supporting a large set of common instructions from the same ISA family and a coherent address space greatly simplifies software compatibility. Second, designers will likely choose a base ISA for all cores and extend some cores with special features, such as wider vector processing, or defeature some to save power. Third, since processors from the same company often already support a large set of common instructions, our model fits naturally and allows companies to re-use products to lower costs.

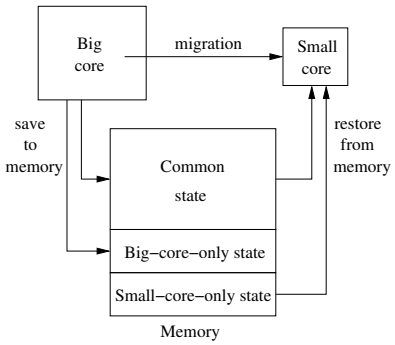


Figure 4: Thread migration from a big core to a small core.

2.2 OS Challenges

Heterogeneous architectures present challenges to both user and system software. Designs such as Cell* [12], CUDA* [23], and EXOCHI [30] run the OS on the "main" cores and access the "extra" cores as coprocessors via libraries or OS drivers. With the overlapping-ISA model, we allow the OS to run on every core and use a single scheduling algorithm. However, since most OSes assume homogeneous processors, they can face two sets of challenges:

Correctness. OSes typically query processor features on the bootstrap processor (BSP) and assume the same for every core. This assumption becomes invalid for heterogeneous processors. With instruction-based asymmetry, software can fail on one core but succeed on another. This needs to be handled properly to ensure correct execution.

Performance. Even when software runs correctly, obtaining high performance can be challenging. With performance asymmetry, an immediate challenge is how applications can share the high-performance cores fairly, especially when they belong to different users. OS scheduling should also enable consistent application performance across different runs. Otherwise, a thread may execute on a fast core in one run but a slow one in another, causing performance variations. Scheduling is further complicated as threads can perform differently on different cores. In general, one would expect higher performance on a faster core; however, for I/O-bound applications, this may not be true. Choosing the right thread-to-core mappings can be challenging.

3 Supporting Instruction-based Asymmetry

To support instruction-based asymmetry, we extend an existing OS with a *fault-and-migrate* mechanism. Our design requires that the hardware generate a fault-type exception when executing an unsupported instruction, which most processors already support. In the fault handler, we migrate the faulting thread to one of the cores that supports the faulting instruction. On the new core, the thread resumes execution and re-executes the faulting instruction.

During a migration, we save in memory the architectural state of the faulting thread on the old core and restore it on the new core, similar to how existing OSes perform thread migration. Since registers on the two cores may differ, we restore only those present on the new core. For registers unique to the old core, we keep their state in memory— if the thread migrates back to the old core or one of the same type, it restores the state. Figure 4 illustrates this design.

Different policies can control when the thread should migrate back. One policy is to migrate back after it has run for a certain amount of time on the new core. To prevent thrashing, a potentially better policy is to migrate back only when the thread has not executed any instruction that could fault again on its original core for a certain amount of time. Instead of its original core, the thread can also migrate back to the least loaded core of the same type as the original core. Section 5 describes our choices in the implementation.

3.1 Hardware Support

Fault-and-migrate requires that the hardware generate a fault-type exception when executing an unsupported instruction. For Intel® Architecture (IA), this exception already exists, known as the invalid opcode exception, or UD fault. However, some applications may trigger "normal" UD faults unrelated to asymmetry via binary re-writing or the `ud2` instruction to trap execution of certain code paths. Such a fault corresponds to an instruction that no core in the system supports. Thus, if relying on UD faults, fault-and-migrate faces two issues. First, it needs to distinguish between "normal" faults and faults that trigger a migration. Second, it needs to identify which cores support the faulting instruction so that the thread can migrate to one of them.

To address these issues, we could provide hardware support. Two types of support can be useful: first, a mechanism for the OS to discover different ISAs in the system and their mappings to cores; second, extended UD fault reporting that specifies which ISA a faulting instruction belongs to. With this information, the OS can construct an ISA-to-core mapping at boot time. On a UD fault, if no ISA is appropriate, the OS treats the fault as "normal"; otherwise, it migrates the thread to a core with the appropriate ISA.

To determine which ISA a faulting instruction belongs to, the instruction decoder of each core needs to understand encodings of every ISA in the processor, which can be costly to implement. If this hardware support is unavailable, the OS could implement a software decoder at the cost of performance. Alternatively, the OS can perform fault-and-migrate conservatively on every UD fault. After the thread migrates to a new core, if it faults again immediately on the same instruction, both ISAs must not support the instruction and the fault is "normal". To improve performance, the OS can cache addresses of instructions that incur "normal" faults to avoid migration if they execute again.

3.2 Discussion

Support for core pinning. Some OSes support core pinning to confine threads to subsets of cores. When deciding where to migrate a thread, we consider only cores that both support the faulting instruction and are allowed by core pinning. If no such core exists, our fault handler sends a signal to the faulting thread, which can choose to either abort or invoke a handler of its own.

Migration versus emulation. Migration between cores with disjoint caches causes a thread to re-load cache state with extra cache misses. Previous work [20] shows that this overhead is negligible on SMP systems, but can be significant on NUMA. We expect a similar trend in future multicore systems. When migration overhead is high, we can emulate faulting instructions instead. If handling the faults is still too costly, we could use binary translation to avoid the unsupported instructions altogether.

Faulting in kernel mode. Support for fault-and-migrate in kernel mode presents two challenges. First, certain code paths such as critical sections are non-preemptible and cannot be transparently migrated. For example, the code may assume that local CPU's run queue is locked; if the thread running this code migrates, the assumption could be invalid. Second, even if the code is preemptible, the faulting instruction might be a privileged instruction that changes the CPU behavior. Migrating such code to a different core transparently would result in the OS incorrectly assuming that the state change occurred in the faulting core. Without hardware support, deciding if a instruction can be safely migrated in kernel mode becomes expensive. For this reason, in this paper, we only support fault-and-migrate in user mode and require that all kernel code only use instructions available on every core. If, besides fault-and-migrate, the rest of the OS also becomes asymmetry-aware, e.g., the OS may select different instructions to run based on core types, then this requirement may become unnecessary.

4 Supporting Performance Asymmetry

A key task for any OS is to balance application performance and system throughput. For this reason, most OSes balance load evenly on each CPU and ensure that each thread receive a a fair share of CPU time. Our design applies the same principle. With performance asymmetry, we balance load proportionally to each CPU's performance. To maximize application performance, we ensure that threads run on high-performance cores whenever they are available. To maximize throughput, we allow each thread to receive a fair share of CPU time on the high-performance cores. If power consumption is of concern, threads may prefer to run on low-power cores. This paper focuses only on performance and we leave power-related studies as future work.

For I/O or memory-bound threads, running on faster cores may not always improve performance. For multithreaded applications, running non-critical threads faster may not improve overall performance. Identifying these scenarios, however, requires detailed knowledge about application characteristics. No production OS assumes this knowledge. Likewise, our design treats all threads equally and assumes that they can obtain higher performance if running on faster cores. With a similar set of design principles, we require only incremental changes to existing OSes, making our design easy to deploy. The rest of this section describes our algorithms to support performance asymmetry.

4.1 Quantifying CPU Performance

An essential component of our algorithms is to assign a performance rating per CPU such that we can estimate performance differences if a thread is to run on different CPUs. There are various ways to obtain CPU ratings. Our design allows the OS to run a simple benchmark of its choice at boot time and set a default rating for each CPU. When the system is up, the OS or user can run complex benchmarks such as SPEC CPU* to override the default ratings if desired. The processor manufacturer can also provide CPU ratings, which the OS can use as the default. All of these approaches produce the same result, i.e., a static rating per CPU. If the rating of a CPU is X times higher than the rating of another CPU, we say this CPU is X times faster.

The user or OS could adjust the ratings dynamically based on workloads. When application performance is stable, the processor may also perform dynamic frequency and voltage scaling, in which case, the OS could assume linear change in application performance and adjust core ratings proportionally to the frequency change. We do not explore these options in this paper. Although static ratings may not accurately reflect performance for all applications, they provide a simple, generic framework. If needed, one can always extend it with more application-specific policies.

4.2 Faster-First Scheduling

Our first algorithm considers the case where the system is lightly loaded with no more threads than the faster CPUs (i.e., CPUs with higher ratings). If two CPUs are idle and a thread can run on both of them, we always run it on the faster CPU. The algorithm consists of two components:

Initial placement. When scheduling a thread for the first time after its creation, if two CPUs are idle, we always choose the faster one to run it. If none is idle, our algorithm has no effect and the OS performs its normal action, typically selecting the most lightly loaded CPU.

Dynamic migration. During execution, a faster CPU can become idle. If any thread is running on a slow CPU, we

preempt it and move it to the faster CPU. Thus, if the total number of threads is less than or equal to the number of faster CPUs, every thread can run on a faster CPU and achieve maximum performance.

If there are more threads than faster CPUs, no one can run on a faster CPU alone. To maximize throughput, it is important that the threads receive fair shares of the CPUs. Next section discusses how we achieve fair scheduling.

4.3 Fair Scheduling

In Section 4.3.1, we provide background on SMP fair scheduling. Section 4.3.2 extends the notion of fairness to performance-asymmetric systems. In Section 4.3.3, we describe an SMP fair scheduling algorithm, and extend it to performance-asymmetric systems in Section 4.3.4.

4.3.1 Background on SMP Fairness

Consider an SMP system with P identical CPUs and N threads. Each thread i, $1 \leq i \leq N$, has a weight w_i, either specified by the user or derived from the thread's priority. A scheduler is perfectly fair if (1) it is work-conserving, i.e., it never leaves a CPU idle if there are runnable threads, and (2) it allocates CPU time to threads in exact proportion to their weights. Such a scheduler is commonly referred to as Generalized Processor Sharing (GPS) [24]. Let $S_i(t_1, t_2)$ be the amount of CPU time that thread i receives in interval $[t_1, t_2]$. A GPS scheduler is defined as follows [24].

Definition 1. *A GPS scheduler is one for which*

$$\frac{S_i(t_1, t_2)}{S_j(t_1, t_2)} \geq \frac{w_i}{w_j}, j = 1, 2, \ldots, N$$

holds for any thread i that is continuously runnable in $[t_1, t_2]$ and both w_i and w_j are fixed in that interval.

A GPS scheduler is idealized since, for Definition 1 to hold, all runnable threads must run simultaneously and be scheduled with infinitesimally small quanta, which is infeasible. In practice, all fair schedulers emulate GPS approximately and are evaluated by their *lag*, which measures their closeness to GPS [5]. An algorithm is considered to have strong fairness if its lag is bounded by a small constant.

4.3.2 Fairness for Performance Asymmetry

In this section, we extend the notion of SMP fairness to performance-asymmetric multiprocessors. Intuitively, the CPU time on a fast core is "worth" more than a slow core. Thus, our idea is to scale CPU time accounting based on each CPU's performance. For example, one unit of time on a two times faster CPU would be equivalent to two units of time on a slow CPU. After obtaining per-CPU ratings, we normalize them to the CPU of the lowest rating. For simplicity, hereinafter, whenever we refer to a CPU rating, we mean its normalized value. Thus, the slowest CPU in the system has a rating of one; the rating of any other CPU indicates how many times faster the OS considers it is.

Let R_p denote the rating of any CPU p. We define the *scaled time* on CPU p at real time t to be $R_p \cdot t$. Thus, if a task runs for x seconds on CPU p, its scaled CPU time is $R_p x$. Let $S_{i,p}(t_1, t_2)$ be the amount of real CPU time that thread i receives on CPU p in time interval $[t_1, t_2]$. Since a thread may migrate to different CPUs during any time period, we define scaled CPU time as follows:

Definition 2. *The scaled CPU time that thread i receives in interval $[t_1, t_2]$ is $\hat{S}_i(t_1, t_2) = \sum_{p=0}^{P-1}(R_p \cdot S_{i,p}(t_1, t_2))$.*

Replacing *CPU time* with *scaled CPU time*, we can extend SMP fairness to performance-asymmetric systems, for which a scheduler is perfectly fair if (1) it is work-conserving, and (2) the scaled CPU time of each thread in any time interval is proportional to its weight. Similarly, a GPS scheduler can be re-defined as follows.

Definition 3. *A GPS scheduler for any performance-asymmetric system is one for which*

$$\frac{\hat{S}_i(t_1, t_2)}{\hat{S}_j(t_1, t_2)} \geq \frac{w_i}{w_j}, j = 1, 2, \ldots, N$$

holds for any thread i that is continuously runnable in $[t_1, t_2]$ and both w_i and w_j are fixed in that interval.

Given these definitions, existing SMP fair scheduling algorithms can apply to asymmetric systems, as long as we change each use of CPU time to scaled CPU time. Prior research [19] shows that distributed weighted round-robin (DWRR) achieves better fairness, performance, and scalability than other existing algorithms. Thus, we choose to extend DWRR to support performance asymmetry. Next, we provide background on the original DWRR algorithm.

4.3.3 Original DWRR Algorithm

DWRR works on top of an existing scheduler using per-CPU thread run queues, a common design in most OSes. It maintains a *round number* per CPU, initially zero. For each thread, DWRR defines its *round slice* to be $w \cdot B$, where w is the thread's weight and B is a constant, *round slice unit*. A round is the shortest time period during which every thread in the system completes at least one of its round slice. A thread's round slice determines its total CPU time allowed in each round. For example, if a thread has weight two and B is 30 ms, then its total runtime per round is at most 60 ms. DWRR consists of two components: *round slicing*, which enables local fairness on each CPU, and *round balancing*, which enables global fairness across CPUs.

Round slicing. Besides the existing run queue per CPU, which we call *round-active*, DWRR adds one more queue,

round-expired. On each CPU, *round-active* and *round-expired* are initially empty and the round number is zero. The scheduler inserts runnable threads into *round-active* and dispatches from there, as it normally does. For all threads in *round-active*, the CPU's round number defines the round in which they are running. DWRR places no control over threads' dispatch order and how long they run once dispatched, which is controlled by the underlying scheduler.

With any existing scheduler, a thread may run for a while, yield to another, and run again. DWRR monitors each thread's *cumulative* CPU time in a round. Whenever it exceeds the thread's round slice, DWRR preempts the thread, removes it from *round-active*, and inserts into *round-expired*. Thus, the invariant is that if a CPU's round number is R, then all threads in its *round-active* queue are running in round R and all threads in *round-expired* have finished round R and are waiting to start round $R+1$. Next, we discuss when a CPU can advance from round R to $R+1$.

Round balancing. DWRR ensures that all CPUs in the common case differ at most by one in their round numbers. This property enables fairness across CPUs because it allows threads to go through nearly the same number of rounds (i.e., run for the same number of their respective round slices) in any time interval. To aid round balancing, DWRR maintains a global variable, $highest$, as the highest round number among all CPUs at any time. Let $round(p)$ be the round number of any CPU p. Whenever p's *round-active* turns empty, DWRR performs round balancing:

Step 1: If $round(p)$ equals $highest$ or p's *round-expired* is empty, then

(i) DWRR scans other CPUs to identify threads in round $highest$ or $highest - 1$ and currently not running (excluding those that have finished round $highest$). These threads exist in *round-active* of a round $highest$ CPU or *round-active* and *round-expired* of a round $highest - 1$ CPU.

(ii) If step i finds a non-zero number of threads, DWRR moves X of them to *round-active* of p, where X is implementation-specific. Note that after all X threads finish their round slices on p, p's *round-active* turns empty again. Thus, it will repeat Step 1 and can potentially move more threads over.

(iii) If step i finds no threads, then either no runnable threads exist or all are running, so p is free to advance to the next round. Thus, DWRR continues to step 2.

Step 2: If p's *round-active* is (still) empty, then

(i) It switches p's *round-active* and *round-expired*, i.e., the old *round-expired* becomes the new *round-active* and the new *round-expired* becomes empty.

(ii) If the new *round-active* is empty, then either no runnable thread exists or all runnable threads in the system are already running; thus, DWRR sets p to idle and $round(p)$ to zero. Else, it increments $round(p)$ by one, which advances all local threads to the next round, and updates $highest$ if the new $round(p)$ is greater.

Finally, whenever the OS creates or awakens a thread, DWRR locates the least loaded CPU among those that are either idle or in round $highest$. It then inserts the thread into *round-active* of the chosen CPU. If this CPU is idle, DWRR sets its round number to the current value of $highest$.

4.3.4 A-DWRR: Extending DWRR to Performance-Asymmetric Systems

A-DWRR extends DWRR to achieve fair scheduling for performance-asymmetric systems. First, we replace each use of CPU time in DWRR with scaled CPU time. The round slice of each thread remains the same. During each round, instead of CPU time, A-DWRR monitors the scaled CPU time of each thread. For example, for two CPUs of ratings one and two, if a thread runs on each CPU for one second, its total scaled CPU time is three. The scaled CPU time progresses at a faster rate on faster CPUs. Intuitively, if a thread spends more time on a faster CPU, it completes more work per unit of time, but also exhausts its round slice more quickly. Thus, A-DWRR can preempt it quickly and move other threads to share the faster CPU fairly.

Figure 5 shows an example with four threads, A, B, C, and D, each of weight one and round slice of one time unit. CPU 0 has a rating of two and CPU 1 has rating one. At time 0, A and B are in *round-active* of CPU 0, and C and D are in *round-active* of CPU 1. At time 1, both A and B have run for 0.5 time units, i.e., 1 unit of scaled CPU time. Thus, they both have completed one round and A-DWRR moves them to *round-expired*. Since *round-active* becomes empty, CPU 0 performs round balancing and moves D over. At this time, the scaled CPU time of both C and D is 0.5. At time 1.25, since D has been running alone for the past 0.25 time units, its scaled CPU time becomes 1. Thus, D moves to *round-expired*. CPU 0 performs round balancing again. Since C is running, CPU 0 finds no thread to move. It switches *round-active* and *round-expired*, and advances to round 1. Now, with A, B, and D all competing on CPU 0, thread C's scaled CPU time will advance faster on CPU 1, even though the CPU is slower. Thus, at some later time, CPU 1 will perform round balancing and move threads from CPU 0. This process continues with threads moving back and forth in a controlled fashion, allowing every thread to receive a fair share of the scaled CPU time. Note that, if only thread C exists on CPU 1 at time 0, then the scaled CPU time of A, B, and C would advance at the same rate. Thus, both CPUs go to the next round at the same time and A-DWRR would trigger no thread migration.

Using only scaled CPU time is insufficient if the system is under-utilized, as Figure 6 illustrates. Assume that all threads have weight one and a round slice of one. In

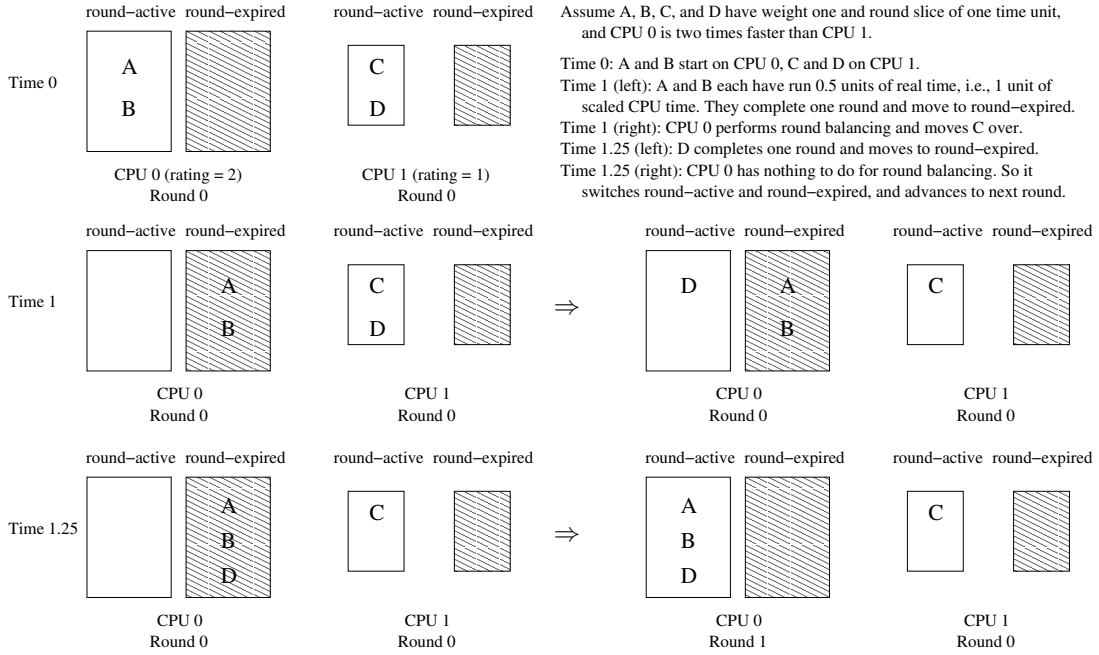
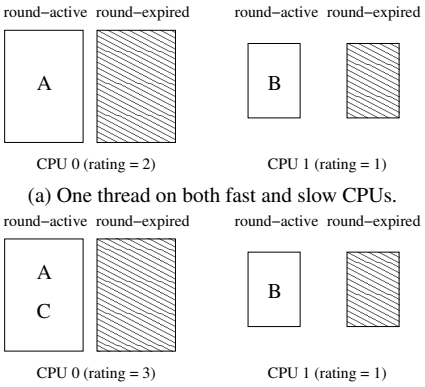


Figure 5: Example of A-DWRR's operation using scaled CPU time.



(a) One thread on both fast and slow CPUs.

(b) Two threads on a 3x faster CPU.

Figure 6: Examples of unfairness with original DWRR. All threads have an equal weight. In both cases, B cannot utilize CPU 0.

Figure 6(a), thread A completes its round slice at time 0.5, while B completes at time 1. In Figure 6(b), both A and C complete their round slices at time 2/3, while B completes at time 1. In both examples, CPU 0's *round-active* turns empty before CPU 1. When it performs round balancing in Step 1(i), CPU 0 can never move B over since it is the only thread on CPU 1 and is always running. Therefore, B never gets a chance to run on the faster CPU, causing it to receive a less amount of scaled CPU time than other threads. Note that, it is not a problem if CPU 0 contains two threads in Figure 6(a) and three in 6(b), in which case, it is fine for thread B to never run on the faster CPU, since each thread does receive an equal amount of scaled CPU time.

In general, the problem occurs when there is at most one thread on each slow CPU and the *round-active* queue of each fast CPU turns empty faster than that of each slow CPU. When both conditions are true, we say the system is *under-utilized*. To solve this problem, we extend DWRR as follows. When CPU p performs round balancing in Step 1(i), it also scans each slow CPU and identifies CPU s for which three conditions are true: (1) $round(s) < round(p)$, (2) $round(s)$ is the minimum among all slow CPUs, and (3) exactly one thread exists on CPU s and is running.

If Step 1(i) finds no thread to move to CPU p, we swap two threads between CPUs p and s. First, we preempt the current running thread, T_1, on CPU s. Second, we select an arbitrary thread, T_2, from *round-expired* of CPU p (whose *round-active* must be empty now). Finally, we move A to *round-active* of CPU p and B to *round-active* of CPU s. Since T_2 was waiting to advance to the next round on CPU p, after moving it to CPU s, we set the round on CPU s to $round(p) + 1$. We do not change the round on CPU p, since other threads on p still rely on it. Instead, after moving thread T_1 to CPU p, we set its round slice to $(1+round(p)-round(s)) \cdot w \cdot B$, where w is p's weight and B is the round slice unit. This gives T_1 a credit such that it can catch up to $round(p)$ quickly. After this first round, we reset T_1's round slice to its normal value $w \cdot B$.

5 Implementation

We used a multiprocessor system to emulate an overlapping-ISA heterogeneous system. Our system contains an Intel® S5000PAL dual-socket board, with a quad-core Intel® Xeon® X5355 processor in one socket and a quad-core E5440 in the other. The E5440 supports SSE4.1 instructions, whereas the X5355 does not. Using proprietary tools, we can adjust each core's frequency, L2 size,

and instruction execution width to create various heterogeneous configurations. We also modified the BIOS to bypass checks that would otherwise prevent the system from booting due to processor asymmetry.

A potential difference between our system and a future heterogeneous processor is that not all cores share a last-level cache, which is typical in today's multi-core designs. Thus, thread migration could be more costly in our system. Nevertheless, this system enables us to obtain insights into the effectiveness of our OS designs in supporting future heterogeneous processors. Next, we discuss our implementation of these designs in the Linux kernel 2.6.24.

5.1 Discovering Asymmetries

Linux uses the CPUID instruction at boot time to enumerate features on each x86 CPU, including SSE4.1. We extend it to construct a bitmap, where a one in bit i indicates SSE4.1 support on CPU i and zero otherwise. Linux also assumes a common timestamp counter (TSC) frequency for all CPUs. It calibrates the TSC only on the BSP at boot time and keeps global variables for the TSC frequency (`cpu_khz`) and cycles per nanosecond (`cyc2ns_scale`). Using the TSC and these variables, the scheduler performs various timing tasks, such as process runtime accounting. In our system, the processors have different TSC frequencies and thus can cause incorrect timing in the scheduler. We modified Linux to calibrate TSC frequency on every CPU and maintain the variables and their operations on a per-CPU basis. To quantify CPU ratings, we run SPEC CPU2006* benchmarks offline for each core configuration and use the sum of the integer and floating-point scores as the core rating. When our system boots, we pass the rating of each core type as a kernel boot-time argument.

5.2 Supporting Instruction-based Asymmetry

When an SSE4.1 instruction executes on one of the big cores, the CPU generates a UD fault.

Fault handling. On a UD fault, Linux sends a SIGILL signal to the user process by default. We modify this fault handler to perform fault-and-migrate. To distinguish from "normal" faults, we assume hardware support to identify faults due to instruction-based asymmetry. For experimentation purposes, our code assumes that every UD fault is due to SSE4.1, which is the case in all of our workloads.

Migrating on a fault. Our fault handler changes the affinity mask of the faulting thread to include only cores capable of SSE4.1 and allowed by its original mask. To retain the fairness properties of A-DWRR, we select a CPU in the highest round among those in the new mask as the destination for the thread, since CPUs in lower rounds are lagging behind and should allow no more threads to run on. To migrate the thread, the fault handler awakens Linux's migration thread on the current CPU and suspends itself (i.e., the faulting thread). When the migration thread runs, it migrates the faulting thread to the chosen destination.

Migrating back. Our first policy migrates the thread back after it has run for T timer ticks on the new core, where T is tunable at run time. Our second policy migrates the thread back only after it has run for T ticks without incurring any SSE4.1 instruction. Using past information, this policy predicts if the thread will fault and migrate again, and thus can potentially prevent the thread from thrashing between the big and small cores. Since existing hardware counters cannot count SSE4.1 instructions, we program them to count all SIMD instructions. However, all of our SSE4.1 benchmarks frequently use SIMD instructions, such as SSE2, even when they do not execute SSE4.1. As a result, conditions in the second policy are rarely satisfied and threads often stay on the new core without migrating back. Thus, we do not consider this policy further and focus on the first policy.

After deciding to migrate a thread back, we restore its affinity mask. To retain fairness for A-DWRR, we migrate the thread to the highest round CPU among those in its affinity mask and of the same type as its original core.

5.3 Supporting Performance Asymmetry

Faster-first scheduling may migrate a *running* thread, for which we must ensure correct migration of the thread's state. We leverage the migration thread mechanism in Linux. When an idle CPU A decides to migrate a running thread on CPU B, it awakens the migration thread on B by sending it an Inter-Processor Interrupt (IPI). Upon receiving the IPI, CPU B immediately saves the state of the running thread, switches it out, and switches the migration thread in. Once running, the migration thread simply moves the thread from the run queue of CPU B to that of CPU A.

We implemented A-DWRR based on the SMP DWRR code at `http://triosched.sourceforge.net`. We replaced each use of CPU time with scaled CPU time based on the SPEC CPU2006* ratings. To achieve fairness when the system is under-utilized, we implemented the extensions as discussed in Section 4.3.4.

6 Evaluation

Table 1 describes our benchmarks. The top three benchmarks are single-threaded and evaluate our support for instruction-based asymmetry. For each of them, we run two versions, one with SSE4.1 and one without. The bottom four are standard benchmarks with no SSE4.1 instructions; we use them to evaluate performance asymmetry.

In addition to the ISA asymmetry (SSE4.1 support), we modify our test system to increase the amount of performance asymmetry. The native X5355 cores are used to em-

Table 1: Description of the benchmarks. The top three are used to evaluate instruction-based asymmetry.

Conv3D: 10 runs of 3D convolution of data size $512 \times 512 \times 512$ [8].
Mandelbrot: Mandelbrot-set map evaluation of 10K×10K data [15].
FFmpeg: Conversion of a 170 MB QuickTime* movie into MPEG1 format using FFmpeg (`http://www.ffmpeg.org`).
SPEC OMP* V3.1: We use the medium version with reference inputs and eight OpenMP threads.
SPECjbb2005* V1.07: We run from 1 to 16 warehouses (threads) and report average throughput of 8 to 16 warehouses.
Kernbench v0.30: We use the parallel make benchmark to compile the Linux 2.6.15.1 kernel source with 32 threads.
x264: We use the x264 video encoder to convert a 744 MB YUV format movie into MP4 with eight threads.

ulate big cores (8 MB L2 cache, 2.66 GHz). Using proprietary tools, we configure the E5440 cores to emulate small cores with a 2 MB L2 cache and narrow execution bandwidth by disallowing execution of instructions on two sets of execution units whenever possible.

6.1 Instruction-based Asymmetry

To emulate the accelerator usage model in Section 1, we configure the small cores with a 2 GHz frequency, resulting in a 32% lower SPEC CPU2006* rating than the big cores.

Fault-and-migrate performance. We perform three experiments for the three instruction-asymmetry benchmarks. First, we run the non-SSE4.1 version by pinning it on a big core, which gives the performance of running on a homogeneous system of big cores without SSE4.1. Second, we run the SSE4.1 version without pinning. With faster-first scheduling, it starts on a big core; on an SSE4.1 instruction, it faults and migrates to a small core and later back to a big core. Thus, the benchmark migrates back and forth between the big and small cores, allowing us to evaluate overheads of fault-and-migrate. As discussed in Section 5.2, fault-and-migrate allows a thread to stay on a small core for T timer ticks before migrating back. To evaluate the impact of T, we repeat this experiment with T equal to 1, 2, 4, and 8, where one tick in our system is 4 ms. Finally, to emulate a costly design of homogeneous big cores *with* SSE4.1, we re-configure each small core to have equivalent performance to the big core. By pinning the SSE4.1 version of each benchmark to this core, we get an upper bound for any heterogeneous configuration with fault-and-migrate.

Figure 7 shows our results, with all data normalized to the non-SSE4.1 case. Above the bar for each T value, we also show its fault rate, i.e., average number of UD faults per second; the migration rate is twice this number, since each fault triggers two migrations. We make the following observations from these results.

With fault-and-migrate, both Conv3D and Mandelbrot outperform the non-SSE4.1 case. Although a small core has a 32% lower rating, fault-and-migrate enables a speedup

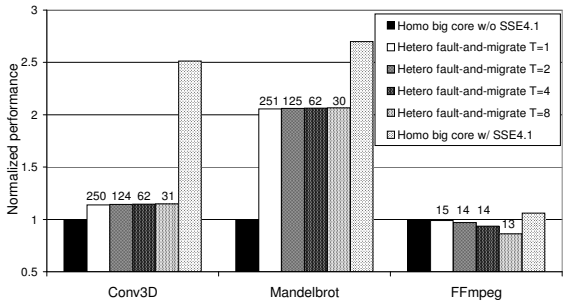


Figure 7: Performance results for fault-and-migrate.

of 1.2 for Conv3D and 2.1 for Mandelbrot over the non-SSE4.1 case. Compared to the ideal case (homogeneous big cores with SSE4.1), our performance is much lower. As we show shortly, fault-and-migrate overhead is nearly zero. The main overhead that contributes to this difference comes from the artifact that, as we configure each small core with a smaller frequency and cache, SSE4.1 performance is reduced as well. Thus, the small core cannot deliver the same SSE4.1 performance as in the ideal case.

For Conv3D and Mandelbrot, larger T values reduce faults, but have no performance impact. When T is one, both have a fault rate of 250, i.e., one fault per timer tick. Thus, each benchmark repeats this pattern: on a fault, it migrates to a small core; after a tick, it migrates back to a big core and faults immediately again. The same is true as T increases. Thus, each benchmark spends nearly no time on a big core, resulting in a constant runtime for the different T values. For FFmpeg, performance decreases as T increases. Comparing its leftmost and rightmost bars reveals that SSE4.1 brings at most a 6% speedup. However, a larger T value causes the benchmark to stay longer on a small core and, consequently, a larger slowdown, which quickly outweighs the benefit of SSE4.1. Another observation is that the fault rate stays nearly constant as T increases, because this benchmark has so few SSE4.1 instructions that although the total number of migrations decreases slightly as T increases, it is too small to change the overall fault rate.

During these experiments, we found that different applications can have different sensitivity to the T value. For example, both Conv3D and Mandelbrot stream through memory and thus thread migrations have less impact on their performance, whereas FFmpeg is more sensitive to migrations. Our code uses one tick as the default T value; future work will explore schemes to adjust it dynamically.

Fault-and-migrate overhead. To prevent performance asymmetry from perturbing our measurements, we configured every core to have equivalent performance to the big core. The system exhibits only instruction-based asymmetry since only one core type support SSE4.1. We ran the above experiments again. Figure 8 shows, for different T values, the slowdown of each benchmark under fault-and-migrate over pinning it on a big core with SSE4.1. With-

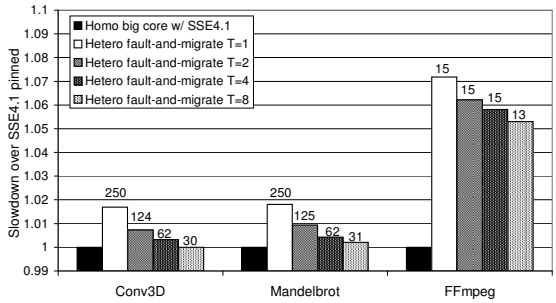


Figure 8: Overhead of fault-and-migrate.

out performance asymmetry, these results indicate the overhead of fault-and-migrate. Above each bar, we also show the corresponding fault rate. For Conv3D and Mandelbrot, as T increases, their number of migrations decreases greatly, so does the fault-and-migrate overhead. For FFmpeg, the number of migrations decreases slightly, but still enough to lower the migration overhead, due to this benchmark's cache access pattern. Finally, when T equals eight ticks, fault-and-migrate incurs zero overhead for Conv3D and Mandelbrot, and a 5.3% slowdown for FFmpeg.

Fault-and-migrate increases thread migrations, which can have different impact on different applications, as we have shown. A migration involves moving a thread from one CPU to another and refilling caches on the new CPU. Li et al. [20] showed that overhead of the latter often dominates. Thus, we focus on it and use a microbenchmark [19] to evaluate cache refill costs on our system with instruction-based asymmetry but no performance asymmetry.

Our results show that, as the program working set size increases, the cost to refill caches on the new CPU increases. However, the cost is very different for migrations between CPUs with and without a shared L2 cache. When the two CPUs have separate caches, we observed migration costs of up to 1.8 ms for working set size of 4 MB and 498 μs for working set size of 512 KB. In contrast, on CPUs with a shared L2, since the program only needs to refill the L1 after migration, the migration cost is much lower with a maximum of only 4.1 μs for 4 MB working set size and 3.7 μs for 512 KB working set size. As the multi-core trend continues, we expect more designs with shared caches and thus low migration costs. Furthermore, our results are conservative—actual cache refill costs can be much lower due to the latency hiding capabilities of hardware prefetching and out-of-order execution in modern processors.

6.2 Performance Asymmetry

We evaluate our algorithms using a configuration with a higher level of performance asymmetry by reducing the small core frequency to 1 GHz, resulting in a 3.4:1 ratio in SPEC CPU2006* ratings between the big and small cores.

We first evaluate A-DWRR's fairness with a microbenchmark of 12 threads, each incrementing an integer in an infinite loop, and use a modified `top` to monitor CPU

PID	PR	NI	%CPU	TIME+	Work
3310	20	0	71	0:29.22	2372
3312	20	0	71	0:30.08	7733
3320	20	0	71	0:29.84	2438
3318	20	0	68	0:29.80	6663
3315	20	0	67	0:29.88	2425
3319	20	0	67	0:29.14	7473
3321	20	0	65	0:29.54	7589
3316	20	0	65	0:29.20	3327
3317	20	0	65	0:30.16	2448
3311	20	0	64	0:29.82	7636
3313	20	0	64	0:29.04	7464
3314	20	0	63	0:29.08	2360

(a) Stock Linux.

PID	PR	NI	%CPU	%SCPU	TIME+	STIME+	Work
3321	20	0	81	146	0:29.98	1:06.03	4997
3314	20	0	80	146	0:30.52	1:05.94	4999
3325	20	0	75	152	0:30.68	1:06.10	5011
3322	20	0	70	149	0:29.10	1:06.18	4999
3318	20	0	69	146	0:29.90	1:06.01	4996
3315	20	0	68	150	0:29.78	1:06.21	5011
3319	20	0	63	147	0:29.74	1:06.02	4993
3320	20	0	62	146	0:30.22	1:06.04	5001
3317	20	0	61	148	0:30.02	1:06.32	5019
3324	20	0	61	150	0:30.26	1:06.18	5010
3323	20	0	57	148	0:28.82	1:06.08	4992
3316	20	0	52	148	0:27.90	1:06.11	4984

(b) Modified Linux.

Figure 9: Snapshots of `top` for 12 threads on four big and four small cores.

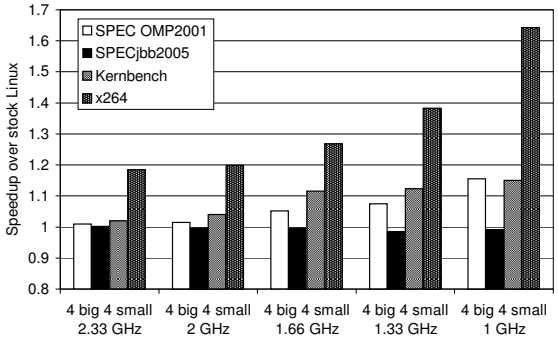


Figure 10: Speedups for various heterogeneous systems.

time allocation. Figure 9 shows our results. The last column shows work done by each thread, where one work unit is a million loop iterations. Figure 9a shows that stock Linux allocates each thread roughly equal CPU time; however, work done by each in the same time interval is drastically different. In Figure 9b, our `top` displays two extra columns: %SCPU shows percent of scaled CPU time each thread receives in every three seconds (default `top` refresh interval); STIME+ shows each thread's cumulative scaled CPU time since start of execution. We see that each thread receives nearly identical scaled CPU time and their work differs at most by 1%, demonstrating A-DWRR's fairness.

We use the bottom four benchmarks in Table 1 to evaluate performance and fairness. Figure 10 shows our speedups over stock Linux for a set of configurations with varying degrees of asymmetry by changing the small core frequency from 2.33 to 1 GHz in 0.33 decrements. All benchmarks except SPECjbb2005* gain higher speedups as the small cores get slower, because faster-first scheduling enables threads to utilize faster cores whenever possible and this benefit increases when the relative speed of a faster core increases. Similarly, A-DWRR enables all threads to

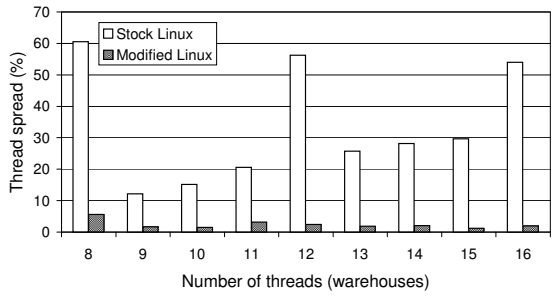


Figure 11: Thread spreads in SPECjbb2005*. Our scheduler achieves strong fairness with a maximum spread of 6%.

progress at a similar pace, preventing any one from being a bottleneck. This benefit also increases as performance asymmetry among the cores widens.

Our scheduler did not improve SPECjbb2005* performance, because this benchmark measures throughput as the total number of transactions, which is not impacted even though some threads monopolize the big cores. However, if each thread represents a different client, this unfairness can seriously impact quality-of-service. SPECjbb2005* defines *thread spread* to be $(max - min)/max$, where max and min are the maximum and minimum number of transactions a thread completes. Figure 11 shows the spreads of stock Linux and our modified one from eight to sixteen total threads. With stock Linux, the maximum spread is 61%, while ours is only 6%, demonstrating much better fairness.

7 Related Work

Prior research assumes either single or disjoint ISAs. Our work lies in between and considers overlapping ISAs, which we believe is more practical. For disjoint ISAs, most manage the "different" cores as coprocessors or peripherals and incur high overhead when moving contexts across address spaces. CUDA* exposes graphics processors as a coprocessor through libraries and OS drivers [23]. Cell* offloads pre-defined code blocks to Synergistic Processor Elements [12] and EXOCHI offloads to a graphics processor via libraries and compiler extensions [30]. These designs place great burden on programmers, whereas we allow the OS to transparently manage all cores as traditional CPUs.

MISP [13, 30] employs proxy execution similar to fault-and-migrate. However, it requires hardware support. Furthermore, any OS service request on the application-managed cores triggers a fault and migration, whereas we incur this overhead only on a missing instruction.

For single-ISA architectures, conventional multiprocessor research [1, 21] has shown that performance-asymmetric designs achieve higher performance than cost-equivalent homogeneous ones. Analytical models by Hill and Marty [14] show that asymmetric multi-core designs provide greater potential speedup than symmetric designs, provided challenges (e.g., scheduling) can be addressed. Traditional graph-based algorithms [4, 27, 28] are impractical due to assumptions such as *a priori* knowledge of task runtime and dependencies. Figueiredo and Fortes [10] studied heterogeneous distributed shared-memory multiprocessors and proposed an algorithm using processor performance ratios, similar to our CPU ratings. Their algorithm, however, is static and not suitable for OS scheduling. Bender and Rabin [6] proposed an algorithm similar to faster-first scheduling, but applied it only to a language runtime.

Recent research has studied heterogeneous multi-core architectures, but none addressed instruction-based asymmetry. Kumar et al. [16, 17] proposed sampling-based scheduling. DeVuyst et al. [9] studied sampling and electron policies adapting to thread execution phases. Bower et al. [7] discussed OS scheduling challenges. All of this work used only simulation. Balakrishnan et al. [3] implemented an algorithm similar to faster-first scheduling, but did not address the fairness issues. Shelepov et al. [26] proposed scheduling using program architectural signatures. Lakshminarayana et al. [18] proposed task size and critical section length aware scheduling. These designs could complement ours to dynamically adjust CPU ratings. Prior power-related studies [2, 11, 22] could also complement our design to help improve both performance and power consumption.

8 Conclusion

Heterogeneous architectures provide a cost-effective solution for improving both single-thread performance and multi-thread throughput. However, they also face significant challenges in the OS design, which traditionally assumes only homogeneous hardware. This paper presents a set of algorithms that allow the OS to effectively manage heterogeneous CPUs. Our fault-and-migrate algorithm enables the OS to transparently support instruction-based asymmetry. Faster-first scheduling improves application performance by allowing them to utilize faster cores whenever possible. Finally, DWRR allows applications to fairly share CPU resources, enabling good individual application performance and system throughput. We have implemented these algorithms in Linux 2.6.24 and evaluated them on an actual heterogeneous platform. Our results demonstrated that, with incremental changes, we can modify an existing OS to effectively manage heterogeneous hardware and achieve high performance for a wide range of applications.

Acknowledgments

We are grateful to Sean McElderry for the BIOS modifications, Srinivas Chennupaty and Avinash Sodani for valuable comments and suggestions, and discussions with Ajay Bhatt, Dan Baumberger, Martin Dixon, Jim Held, Ganapati Srinivasa, Hong Wang, and Youfeng Wu.

References

[1] J. B. Andrews and C. D. Polychronopoulos. An analytical approach to performance/cost modeling of parallel computers. *Journal of Parallel and Distributed Computing*, 12(4):343–356, Aug. 1991.

[2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 298–309, June 2005.

[3] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[4] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):319–330, Apr. 2004.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[6] M. A. Bender and M. O. Rabin. Scheduling Cilk multithreaded parallel programs on processors of different speeds. In *Proc. of the Twelfth ACM Symposium on Parallel Algorithms and Architectures*, pages 13–21, July 2000.

[7] F. A. Bower, D. J. Sorin, and L. P. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28(3):17–25, 2008.

[8] Z. Danovich. 16-bit 3D convolution: SSE4 and OpenMP implementation on the Penryn CPU. *Intel Software Network*, Feb. 2008.

[9] M. DeVuyst, R. Kumar, and D. M. Tullden. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *Proc. of the 20th International Parallel and Distributed Processing Symposium*, Apr. 2006.

[10] R. J. O. Figueiredo and J. A. B. Fortes. Impact of heterogeneity on DSM performance. In *Proc. of the Sixth International Symposium on High-Performance Computer Architecture*, pages 26–35, Jan. 2000.

[11] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd Conference on Computing Frontiers*, May 2005.

[12] M. Gschwind. The Cell* broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3), June 2007.

[13] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple instruction stream processor. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, June 2006.

[14] M. Hill and M. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[15] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, Nov. 2007.

[16] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[17] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 64–75, June 2004.

[18] N. Lakshminarayana, S. Rao, and H. Kim. Asymmetry aware scheduling algorithms for asymmetric multiprocessors. In *Proc. of the Fourth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2008.

[19] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proc. of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Feb. 2009.

[20] T. Li, D. Baumberger, D. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proc. of the 2007 ACM/IEEE Conference on Supercomputing*, Nov. 2007.

[21] D. Menascé and V. Almeida. Cost-performance analysis of heterogeneity in supercomputer architectures. In *Proc. of the 1990 International Conference on Supercomputing*, pages 169–177, June 1990.

[22] 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*, 28(3):26–41, 2008.

[23] NVIDIA. *NVIDIA CUDA* Programming Guide, Version 1.1*. NVIDIA Corporation, Nov. 2007.

[24] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[25] R. Ramanathan, R. Curry, S. Chennupaty, R. L. Cross, S. Kuo, and M. J. Buxton. Extending the world's most popular processor architecture. White Paper, Intel Corporation, 2007.

[26] D. Shelepov, J. C. S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. HASS: A scheduler for heterogeneous multicore systems. *Operating Systems Review*, 43(2):66–75, Apr. 2009.

[27] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, Feb. 1993.

[28] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar. 2002.

[29] M. Venkatachalam, P. Chandra, and R. Yavatkar. A highly flexible, distributed multiprocessor architecture for network processing. *Computer Networks*, 41(5):563–586, Apr. 2003.

[30] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proc. of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 156–166, June 2007.