Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures

Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn
Systems Technology Lab
Intel Corporation
{tong.n.li,dan.baumberger,david.a.koufaty,scott.hahn}@intel.com

ABSTRACT

Recent research advocates asymmetric multi-core architectures, where cores in the same processor can have different performance. These architectures support single-threaded performance and multithreaded throughput at lower costs (e.g., die size and power). However, they also pose unique challenges to operating systems, which traditionally assume homogeneous hardware. This paper presents AMPS, an operating system scheduler that efficiently supports both SMP- and NUMA-style performance-asymmetric architectures. AMPS contains three components: asymmetry-aware load balancing, faster-core-first scheduling, and NUMA-aware migration. We have implemented AMPS in Linux kernel 2.6.16 and used CPU clock modulation to emulate performance asymmetry on an SMP and NUMA system. For various workloads, we show that AMPS achieves a median speedup of 1.16 with a maximum of 1.44 over stock Linux on the SMP, and a median of 1.07 with a maximum of 2.61 on the NUMA system. Our results also show that AMPS improves fairness and repeatability of application performance measurements.

1. INTRODUCTION

Multi-core architectures are becoming mainstream in both server and desktop processors. Over the next decade, we expect to see processors with tens and even hundreds of cores on a chip [6]. To efficiently utilize chip real-estate, recent research [2, 3, 11, 13, 14, 15] advocates performance-asymmetric (or heterogeneous) architectures, where a processor contains multiple cores with the same instruction set but different performance characteristics (e.g., clock speed, issue width, in-order vs. out-of-order). These architectures provide cost-effective platforms for both throughput-oriented applications and applications that demand single thread performance. Prior research [2, 13, 15] demonstrated that, compared to homogeneous ones, asymmetric architectures deliver higher performance at lower costs in terms of die area and power consumption. Besides these architectures by design, performance asymmetry can also arise in homogeneous multi-core systems when cores dynamically switch power states or disable failing components for fault tolerance. To effectively support these architectures, the operating system (OS) must take into account hardware asymmetry when making scheduling decisions. However, OS schedulers traditionally assume homogeneous hardware and do not directly work well on asymmetric architectures.

This paper presents *AMPS*, an asymmetric multiprocessor scheduler that efficiently supports both SMP- and NUMA-style performance-asymmetric architectures. AMPS focuses on three metrics: performance, fairness, and repeatability of application performance measurements. There are three components in AMPS:

(1) *Asymmetry-aware load balancing* ensures that the load on each core is proportional to its computing power.
(2) *Faster-core-first scheduling* ensures that threads run on faster cores whenever they are under-utilized.
(3) *NUMA-aware migration* dynamically predicts thread migration overheads using memory resident sets and controls migration on NUMA-style architectures (e.g., due to cores connected to different memory controllers).

We have implemented AMPS in the Linux 2.6.16 kernel. To emulate future SMP- and NUMA-style asymmetric architectures, we modulate core duty cycles on a four dual-core processor SMP and a 32-processor NUMA system. We evaluate AMPS with SPEC OMP[*], SPECjbb2005[*], Kernbench (parallel make), and Ogg Vorbis (audio encoding). Compared to stock Linux, AMPS achieves a median speedup of 1.16 (average 1.15) on the SMP, and a median of 1.07 (average 1.18) on the NUMA system. Furthermore, our results show that AMPS improves fairness and repeatability of application performance measurements.

The remainder of this paper is organized as follows. In Section 2, we describe the general design of AMPS. In Section 3, we discuss our Linux implementation of AMPS. We present evaluation results in Section 4, discuss related work in Section 5, and conclude in Section 6.

2. SCHEDULING FOR PERFORMANCE-ASYMMETRIC ARCHITECTURES

In this section, we discuss the design space of OS scheduling for performance-asymmetric architectures, our focus, and the AMPS design.

2.1 Design Space

We focus on designing a general-purpose scheduler that enables good performance for most applications. Most existing multiprocessor OSes, such as Windows Server 2003[*], Solaris 10[*], Linux 2.6[*], and FreeBSD 5.2[*], use a distributed run-queue model, where the scheduler maintains one run queue per core; periodically, it balances the load on every core. Compared to a centralized run-queue model, the distributed model achieves higher scalability. Thus, we designed AMPS based on this model.

SC07 November 10–16, 2007, Reno, Nevada, USA
Copyright 2007 ACM 978-1-59593-764-3/07/0011 ...$5.00.

We classify scheduling policies into *thread-dependent* policies, which schedule threads based on their application types and relations to other threads, and *thread-independent* policies, which schedule threads independently regardless of application types and dependencies. Each policy can also have different optimization metrics, such as performance, fairness, and power efficiency.

Most general-purpose OSes use thread-independent policies and favor threads to cores that are less loaded to facilitate load balance or cache-warm to exploit cache affinity. Thread-dependent policies mostly exist in research. Zheng and Nieh [25] dynamically detect process dependencies to guide scheduling. Recent research [9, 20, 22] co-schedules threads with little contention on shared resources, such as caches and functional units. On performance-asymmetric architectures, the design space for thread-dependent policies is even larger. For example, in a system with many in-order cores and a few out-of-order cores, to maximize performance, a scheduler may assign threads of a throughput-oriented application to the in-order cores and single-threaded applications to the out-of-order cores. To conserve power, however, the scheduler may favor in-order cores, though potentially at the cost of performance.

Our design for AMPS focuses on thread-independent policies and the following metrics:

- *Performance*: most applications should achieve good performance.
- *Fairness*: threads with the same priority should receive about the same share of core processing power.
- *Repeatability*: different runs of an application under similar conditions should have similar performance.

Our choices are similar to those of existing general-purpose OSes. However, existing schedulers do not directly work well on asymmetric hardware. For example, without considering hardware asymmetry, they could dispatch a thread to a higher performance core in one run, but to a lower performance core in another run, causing non-repeatable performance results. By having a similar set of design principles, AMPS requires only simple changes to existing OSes, making it easy to deploy. We demonstrate this in Section 3 by showing how to implement AMPS with simple changes to Linux. Our future work will explore other areas of the design space, e.g., policies that take into account cache contention. The rest of this section discusses in detail the components of AMPS.

2.2 Asymmetry-Aware Load Balancing

Conventional OSes perform periodic load balancing to enable threads to share cores fairly. AMPS extends this approach by maintaining the load on each core proportional to its computing power.

Quantifying Core Computing Power. The traditional approach to quantify computing power is to run benchmarks such as SPEC INT[*] and FP[*], and obtain metrics such as instructions-per-cycle (IPC) or million-instructions-per-second (MIPS). The OS can use this approach at boot time to obtain a computing power value for each core. To account for events such as dynamic voltage scaling, the OS can re-evaluate computing power whenever these events occur. We define a core's *scaled computing power*, P, to be its computing power divided by the system's minimum core computing power. For this paper, we have emulated asymmetric systems in which cores differ in frequency. For such systems, we approximate computing power using core frequencies instead of benchmark measurements. Nevertheless, our algorithms are applicable to other types of asymmetric systems as well.

At boot time, AMPS sets the scaled computing power of the core with the lowest frequency to one and any core with a F times higher frequency to $F \times S$, where S is a less-than-one scaling factor, reflecting the fact that an F times higher frequency often leads to less than F times higher application performance since the memory system remains non-scaled. The value of S is a function of core frequency because the rate at which application performance improves levels off as frequency increases. To determine S for different frequencies, AMPS could use empirical data or run benchmarks similarly to what we described above. We have chosen a simplified approach by setting S to be a constant and leave the investigation of more sophisticated approaches as future work.

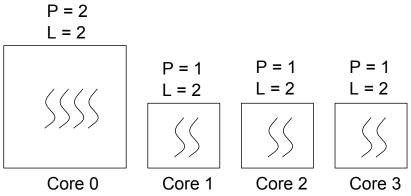


Figure 1: Example for asymmetry-aware load balancing. P is scaled computing power and L is scaled load.

Scaled Load and Load Balancing. Conventional OSes define the load of a core to be the number of threads in its run queue, i.e., run queue length (sometimes scaled by a constant factor). For any core with scaled computing power P, we define its *scaled load*, L, to be its run queue length divided by P. Let L_{max} and L_{min} be the maximum and minimum scaled load of a core in an asymmetric system. We define that the system is load-balanced if $L_{max} - L_{min} \leq 1$. Figure 1 illustrates these definitions with one fast core ($P = 2$) and three slow cores ($P = 1$). The fast core has four threads and each slow core has two. Thus, the scaled load of each core is two and the system is load-balanced. This design enables threads to share cores fairly. In a homogeneous system, conventional load balancing enables threads of the same priority to receive about the same share of CPU time. In our asymmetric model, one unit of time on a core with $P = x$ is equivalent to x units of time on a core with $P = 1$. Thus, by balancing threads proportionately to core computing power, AMPS facilitates fairness.

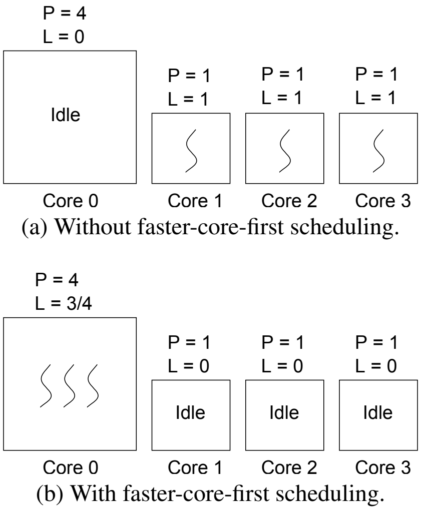
2.3 Faster-Core-First Scheduling

Faster-core-first scheduling enables threads to run on more powerful cores whenever they are under-utilized (i.e., $L < 1$), which load balancing alone cannot achieve in an asymmetric system.

2.3.1 Motivating Example

Figure 2(a) shows a system with one idle fast core ($P = 4$) and three slow cores ($P = 1$) each running one thread. By definition, the system is load-balanced. However, other balanced configurations could lead to better performance. We consider two options: (1) moving one thread to the fast core so that it runs 4 times faster, and (2) moving all three threads to it so that each could run 4/3 times faster (assuming no cache contention). With the first option, after the thread finishes on the fast core, the scheduler moves another thread to it. Assuming zero migration overhead and that all threads perform equal work, our analysis shows that it could lead to 32% smaller total runtime than the second approach. However, when threads have different amounts of work, a long running one could monopolize the fast core and prevent others from sharing it, causing lower system throughput.

To facilitate fairness, AMPS could periodically dispatch a different thread to the fast core based on thread priorities. This approach, however, is complicated and expensive (e.g., it would require a global ordering of all threads). Thus, AMPS adopts the second option, i.e., threads move to and run on a faster core as



(a) Without faster-core-first scheduling.

(b) With faster-core-first scheduling.

Figure 2: Example for faster-core-first scheduling. P denotes scaled computing power and L denotes scaled load.

long as the core is under-utilized. With this approach, a fast core of scaled computing power P behaves like P slow cores; in effect, we have transformed an asymmetric system into a symmetric one, which greatly simplifies the fairness support.

2.3.2 *Faster-Core-First Algorithm*

Initial thread placement. For a newly created thread, schedulers for homogeneous systems often start it on the least loaded core. In AMPS, we compute the new scaled load for each core assuming that the thread would run on it and choose the core with the minimum new scaled load to start the thread; if tied, we choose the faster core. This design ensures that a new thread runs on a faster core if the core is under-utilized. In Figure 2(b), assuming that all cores are initially idle, AMPS dispatches each new thread to the faster core until there are three on it. One more new thread goes to a slow core because otherwise four threads would share the faster core and each would perform similarly to running on a slow core. By dispatching the fourth thread to a slow core, the existing three threads can remain potentially 4/3 times faster. Furthermore, running three threads on the faster core, as opposed to one or two, allows more threads to benefit from the faster core, leading to higher system throughput.

Dynamic thread migration. AMPS's asymmetry-aware load balancing naturally enables threads to migrate to faster cores when they are under-utilized. We further extend it by allowing threads to migrate to cores that have lower scaled load even if their original cores can become idle, which conventional OSes such as Linux perform only in special cases. The benefit of allowing such migrations in the common case is that threads always run on faster cores whenever they are under-utilized. AMPS does not allow a thread to migrate if the scaled load on the destination core after the migration can become greater than the scaled load on the original core before the migration. This condition prevents a thread from migrating if its performance is not likely to improve.

2.3.3 *Discussion*

When a thread migrates to a new core, it incurs compulsory cache misses. Thus, OS schedulers generally avoid migrations unless the system load is significantly unbalanced. AMPS employs the same approach. However, compared to OSes such as Linux, faster-core-first scheduling more frequently allows a *running* thread to migrate, which can introduce extra overheads. First, it can cause more thread migrations. Second, migration of a running thread introduces higher costs than migration of a waiting thread (i.e., a ready thread awaiting its turn to run). Our experience (Section 4.2) shows that the overhead associated with a thread's migration is negligible in SMP systems but can be significant in NUMA systems. Thus, we extend AMPS with NUMA-aware migration policies.

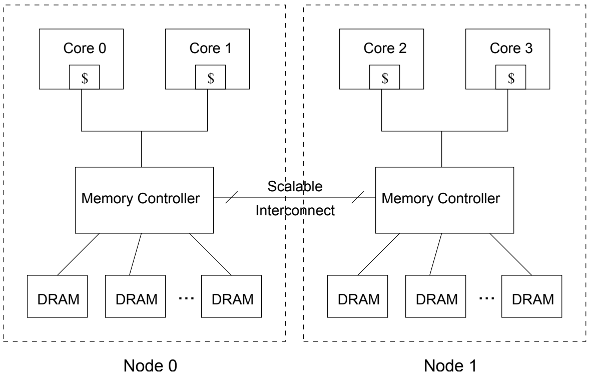


Figure 3: A four-core NUMA system.

2.4 NUMA-Aware Thread Migration

A multi-core NUMA system is similar to a traditional multiprocessor NUMA system except that some cores reside on the same die or in the same package. In Figure 3, we show a conceptual diagram of a four-core NUMA system. Following traditional NUMA terminology, we say that cores connected to the same memory controller are in the same *node*. Within the same node, every core is equidistant to local memory.

2.4.1 *Understanding Migration Overhead*

To understand migration overhead, we consider a running thread that migrates from core A to core B. The migration introduces both software and hardware overhead. The software overhead includes the time to move the thread from A to B, involving four steps: (1) acquiring run queue locks for both cores, (2) dequeuing the thread from core A's run queue, (3) enqueuing it into core B's run queue, and finally (4) releasing the run queue locks. Among these steps, step 1 is the only one potentially expensive. However, since contention on these locks is typically low and the critical section (steps 2 and 3) is short, the overhead of these steps is mostly negligible. Our evaluation in Section 4.2 confirms this observation in Linux.

The hardware overhead has two aspects: (1) the thread incurs extra compulsory misses in cache-type structures, including instruction and data caches and TLBs, after the migration, and (2) penalties of missing in these structures can be higher after the migration due to NUMA. Among these structures, cache misses often dominate and thus we focus on them. In an SMP system, all cores are equidistant to memory and have the same last-level cache (LLC) miss penalty. Therefore, the cache miss overhead associated with a migration primarily comes from compulsory misses on the destination core. In our experience (Section 4.2), this overhead is far smaller than the performance benefits of AMPS in SMP systems. In a NUMA system, the LLC miss penalty can be different on cores in different nodes. For example, suppose that a thread running on core A accesses data in local memory. After it migrates to core B in another node, each LLC miss requires a remote memory access.

In this case, not only do the extra compulsory misses introduce overheads, but all cache misses that occur regardless of whether the thread migrates or not might experience higher penalties after the migration.

Qualitatively, a thread's migration overhead is high if the following conditions are both true:

(1) The thread incurs a high number of LLC misses after the migration.
(2) A large fraction of these LLC misses require remote memory access.

These conditions are not necessary conditions for high migration overhead, but they are sufficient conditions that enable us to detect the common case.

2.4.2 *Predicting Migration Overhead*

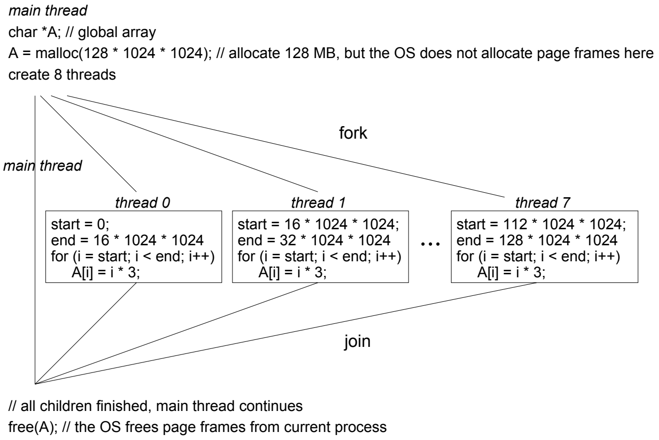
When the OS chooses a thread to migrate, if it can predict the migration overhead, then it can determine if the migration is likely beneficial or not. It is possible to design a range of prediction algorithms at different levels of accuracy. We leave the full exploration of this design space as future work. As a starting point, this paper focuses on a simple algorithm that predicts the overhead of a migration to be either high or low.

Overview. Our algorithm uses thread *working sets* to help examine the two conditions in Section 2.4.1. The intuition is that before a thread migrates, if its working set is larger than the LLC on the destination core, then its number of LLC misses is likely high after the migration (condition 1). To predict if these misses require many remote memory accesses (condition 2), we define *per-thread, per-node working sets*: the working set of thread T on node N is the set of pages that belong to the working set of thread T and physically reside on node N. For two nodes N_1 and N_2, if thread T's working set on N_1 is larger than its working set on N_2, then, probabilistically, it is more likely to access pages backed by N_1. Thus, if a thread migrates to a node on which its working set is relatively small, we predict that it will require a high number of remote memory accesses after the migration.

A key requirement of our algorithm is to keep track of the working set of each thread on each node. We consider two approaches and their tradeoffs.

Working sets. The first approach leverages existing OS support for process working sets. Microsoft Windows* keeps track of the working set of each process. Unix systems, such as FreeBSD*, Linux*, and Solaris*, maintain system-wide page Least Recently Used (LRU) lists, which, in effect, approximate a global working set for all processes. Figure 4 illustrates the differences between process and thread working sets. In this example, a process creates eight child threads, each accessing a different portion of a global array. For each child thread, its working set is part of the sub-array that it accesses, while the working set of the process can be the entire array.

For systems that already support process working sets, we can modify them to track per-thread, per-node working sets as well. Whenever the OS adds a page to a thread's working set, we check from which node the page frame is allocated. In this way, we can track per-node working sets for each thread. For systems that support only global working sets, we can first modify them to support process working sets, and then use the above approach to track per-thread, per-node working sets. Implementing this design, however, can be complicated as it requires major changes to the OS memory management code. To avoid this burden, AMPS adopts a resident-set-based design that is less accurate but much easier to implement.



```
char *A; // global array
A = malloc(128 * 1024 * 1024); // allocate 128 MB, but the OS does not allocate page frames here
create 8 threads
```

```
start = 0;
end = 16 * 1024 * 1024
for (i = start; i < end; i++)
    A[i] = i * 3;
```

```
start = 16 * 1024 * 1024;
end = 32 * 1024 * 1024
for (i = start; i < end; i++)
    A[i] = i * 3;
```

```
start = 112 * 1024 * 1024;
end = 128 * 1024 * 1024
for (i = start; i < end; i++)
    A[i] = i * 3;
```

```
// all children finished, main thread continues
free(A); // the OS frees page frames from current process
```

Figure 4: An example multithreaded program.

Resident sets. Our design leverages existing OS support for tracking process resident sets. The *resident set* of a thread (or process) includes its pages that are currently in memory, i.e., resident. Similar to per-node working sets, we define the resident set of a thread on node N to be the set of pages that belong to the resident set of the thread and physically reside on node N. Since tracking resident sets is much easier than working sets, AMPS tracks the resident set of each thread on each node, which approximates per-thread, per-node working sets.

Most existing OSes maintain the *resident set size* (RSS) for each process. Typically, the OS maintains a per-process RSS counter. Whenever the OS allocates (or deallocates) page frames to (or from) a process, it increments (or decrements) the process's RSS counter. AMPS extends this mechanism to track per-thread, per-node resident sets. It maintains an array of RSS counters for each thread. The size of the array equals the number of NUMA nodes in the system. The i^{th} entry keeps the number of pages that belong to the thread's resident set and physically reside on node i. Whenever the OS allocates (or deallocates) a page frame to (or from) a process, AMPS identifies the thread that triggered this action. For example, if the allocation is due to a page fault, it identifies the faulting thread. It also identifies the node to which the page frame belongs. Finally, AMPS updates the RSS counter corresponding to the thread and the node.

To illustrate a special scenario that we need to address, let us revisit the example in Figure 4. Initially, the main thread allocates a 128 MB global array A. With demand paging, no actual page allocation occurs at this time. The thread then creates eight threads, all with the same address space as the main thread. The child threads each write to a different 16 MB portion of array A. These writes trigger page faults and cause the OS to allocate page frames. Hence, AMPS increments the child threads' RSS counters. After the child threads exit, the main thread frees array A, causing the OS to deallocate A's page frames from the main thread's address space. Consequently, AMPS decrements the main thread's RSS counters by large amounts, which is incorrect because these page frames in fact belonged to the resident sets of the child threads.

To handle this scenario, we associate an *owner list* with each page frame. Whenever the OS allocates a page frame to a process, we add the triggering thread ID to the frame's owner list. When the OS deallocates a page frame from a thread, A, we scan the frame's owner list. For each owner thread, B, if it has already exited, we remove it from the owner list; otherwise, we check if threads A and

B belong to the same process (i.e., share the same address space). If so, we remove thread B from the owner list and decrement its RSS counter by one for the corresponding node. Scanning an owner list could be costly if the list is long. However, the list has multiple entries only if multiple processes share the same page frame; in the common case, it contains only one entry.

Prediction algorithm. When a thread T migrates from core A to core B, our algorithm predicts the migration overhead to be high if *all* of the following conditions are true:

(1) Core A and core B are in different nodes.
(2) Core A is in a node for which thread T has the maximum RSS counter value compared to all other nodes.
(3) The RSS counter value of thread T for core A's node is greater than the LLC size of core B.

If these conditions are true and thread T migrates, it will likely incur a large number of LLC misses and remote accesses to memory in core A's node.

2.4.3 Running Thread Migration Policies

This section discusses three migration policies, including two simple ones and one based on the above resident-set design. These policies are not limited to asymmetric platforms; in fact, they are applicable whenever an OS needs to decide whether or not to migrate a thread. In this paper, we apply these policies only to control the migration of running threads on asymmetric architectures. Whenever the faster-core-first algorithm in Section 2.3 chooses a running thread to migrate, AMPS decides whether to allow the migration using one of the following policies:

- *The Always policy*: With this policy, AMPS always allows a running thread to migrate regardless of the potential overhead. This is the default policy in AMPS for SMP systems. The following two NUMA-specific policies behave equivalently to this policy in SMP systems.
- *The Same-Node policy*: With this policy, AMPS allows a running thread to migrate only within the same node. It is conservative in that it prevents a thread from migrating to a remote, faster core even when the actual migration overhead might be small.
- *The RSS policy*: This policy uses the resident-set-based prediction algorithm. It disallows a running thread to migrate if either it predicts the migration overhead to be high, or the thread is in the memory allocation phase.

The latter condition in the RSS policy prevents a thread from migrating before its resident set stabilizes. While allocating memory, a thread's resident set is small and its predicted migration overhead is likely small. Thus, without this condition, a thread could migrate to other nodes where it continues to allocate memory, causing its resident set to spread across nodes. This would lead to unnecessary migrations and off-node memory accesses.

To detect if a thread is in the memory allocation phase, we use the following heuristic. At each timer interrupt, we check the current RSS (aggregate over all nodes' RSS counters) of the interrupted thread. If it is greater than the thread's previous RSS, we consider the thread in the allocation phase; otherwise not. We then update the thread's previous RSS with the value of its current RSS. This heuristic is effective for our workloads, but we plan to improve it with more sophisticated designs in our future work.

2.4.4 Discussion

The key advantages of our resident-set-based design are ease of implementation and low overhead. It requires only simple extensions to existing OSes, but is effective as we see in Section 4.3. It is also extensible to non-uniform cache access (NUCA) architectures as they share some similar behavior to NUMA. On the other hand, our design has limitations since a thread's resident set only approximates its actual working set. The following list explains the potential inaccuracies in our design.

Table 1: System configurations.

Features	SMP	NUMA
Nodes	1	8
Processors/node	4	4
Cores/processor	2	1
Total cores	8	32
L1 D-cache/core	16 KB	16 KB
L2 cache/core	1 MB	1 MB
L3 cache/core	none	8 MB
L4 cache/node	none	256 MB (DDR2)
Memory/node	8 GB	8 GB
Total memory	8 GB	64 GB
Memory latency	79 ns	local: 68 ns 1-hop: 146 ns 2-hop: 176 ns

- *Stale pages*: the resident set of a thread can include stale pages that are not in its current working set. For example, a thread may access some pages briefly and not touch them any more until finally freeing them. Such stale pages cause the RSS counters in our design to be larger than the actual working set sizes.
- *Shared pages*: when multiple threads in a process share data, their working sets overlap. However, their resident sets do not overlap because, for any shared page frame, only one thread in the process can trigger its allocation and account for it in the thread's RSS counters. Thus, some threads may have RSS counters smaller than their actual working set sizes.
- *Small memory*: a thread's resident set can be smaller than its working set due to limited physical memory.

The first two scenarios could cause AMPS to make migration decisions that negatively affect performance. The last one, however, has negligible impact because, in this case, page faults dominate performance, regardless of what migration decisions AMPS makes. Barring these limitations, evaluation in Section 4.3 shows that our design is effective in all but one of our tests.

3. IMPLEMENTATION

We have implemented AMPS in the Linux 2.6.16 kernel. In this section, we describe how we emulate performance-asymmetric systems and implement the three components of AMPS.

Emulating Performance Asymmetry. To emulate future multicore architectures, we use an SMP system with four dual-core Intel® Xeon™ 7020 processors and an IBM* x460 NUMA eServer* with 32 Intel® Xeon™ 3.3 GHz processors. The NUMA system consists of eight nodes (chassis) that collectively form a 32-processor system. Both systems have no shared caches between cores and hyperthreading disabled. Table 1 shows our system details. The memory latency row shows average latencies on an unloaded system, which we obtained via a microbenchmark. One of the primary differences between our systems and future multi-core systems is that future systems will have better memory subsystems due to shared caches and lower memory latencies. Nevertheless, we believe that our evaluation results are applicable to future systems and provide insights on the effectiveness of AMPS.

To emulate performance asymmetry, we use clock modulation to adjust core duty cycles, which logically adjusts core frequencies. We enable Linux's clock modulation support for 64-bit processors by modifying a kernel configuration file and extending the `pentium4_get_frequency()` function to support our specific processor types. We use this approach for emulation, but, for actual systems, a more general approach would be to have hardware expose the asymmetry attributes, e.g., via the `cpuid` instruction.

Asymmetry-Aware Load Balancing. As we discussed in Section 2.2, one of the parameters that determines the scaled computing power of a core is the scaling function S. For simplicity, we choose S to be a constant of 0.85. Wherever Linux computes or refers to the load of a core, we change it to use the scaled load as discussed in Section 2.2. In this way, simply using Linux's existing `load_balance()` and `load_balance_newidle()` functions enables us to achieve asymmetry-aware load balancing.

Faster-Core-First Scheduling. When a thread calls the `fork` or `exec` system call, Linux uses `find_idlest_group()` to find the least loaded scheduler group, from which it chooses the core to start the child thread (`fork`) or the caller thread (`exec`). In AMPS, we modify `find_idlest_group()` to implement the algorithm in Section 2.3. At each timer interrupt, Linux checks if load balancing is necessary. If so, it invokes `load_balance()`. Similarly, it invokes `load_balance_newidle()` when a core is becoming idle. Both functions call `find_busiest_group()` to find the most loaded scheduler group, from which other groups pull threads. We modify `find_busiest_group()` in two ways. First, we consider a slower core to be more loaded when two cores have the same scaled load. Second, as we discuss next, we allow a running thread to migrate even if its original core can become idle.

The common path of Linux load balancing considers migration of non-running, ready threads. Such a migration simply moves a thread from its current run queue to a different one. However, for a *running* thread to migrate, it needs to save its execution state, move to a different core, restore the state, and finally resume execution. Fortunately, Linux already provides a mechanism to achieve these requirements. On each core, it runs a *migration thread*, which allows the kernel to force a thread to migrate (e.g., when changing a thread's affinity mask).

In our implementation, when core A pulls a thread T running on core B, it records core A as the migration target and calls the `wake_up_process()` function in Linux to awaken the migration thread on core B. This function enqueues the migration thread into core B's run queue and sends an Inter-Processor Interrupt (IPI) to core B such that it immediately switches thread T out and the migration thread in. While running, the migration thread simply moves thread T from the run queue of core B to that of core A.

One complication arises when multiple cores pull a thread at the same time. In this case, we want only one core to succeed. Thus, we associate a flag with each core. Initially, this flag is zero. When core A needs to pull a running thread on core B, it does an atomic compare-and-exchange (`cmpxchg`) on core B's flag: if the flag is zero, it sets it to one; otherwise, the flag remains unchanged. When multiple cores perform this operation at the same time, only one succeeds and awakens the migration thread on core B. When the migration thread completes, it resets core B's flag to zero.

NUMA-Aware Thread Migration. For each process, Linux keeps two counters for the pages in its resident set: `file_rss` and `anon_rss`. The former counts file-backed pages, such as those in a file's memory mappings; the latter counts anonymous pages that have no image on disk, such as those of a user-mode heap or stack. To track per-thread, per-node RSS, we add two RSS counter arrays, `file_rss_per_node` and `anon_rss_per_node`, to each thread's `task_struct`. Wherever Linux updates `file_rss` or `anon_rss`, we perform the same update in the corresponding RSS array. Linux conveniently provides the `page_to_nid()` function that returns for each page the ID of the node where the corresponding page frame resides. Using this function, we can easily identify the right node counter in the RSS array to update.

4. EVALUATION

This section discusses our evaluation methodology and results.

4.1 Methodology

For both our SMP and NUMA systems described in Section 3, we modulate three fourths of the total cores to run at 50% of their full duty cycles, logically making their frequencies 50% lower than the rest of the cores. We believe that this configuration is representative of future platforms, where the number of fast cores is typically fewer than the number of slow cores. Table 2 describes our benchmarks. For SPEC OMP* and Kernbench, we run each benchmark three times and report the average runtime. For SPECjbb2005*, we run it once and report the official metric—the average throughput from N to $2 * N$ warehouses. To evaluate fairness, we run 32 Ogg Vorbis encoder processes on the SMP system and, whenever a process exits, we start another to keep the system over-subscribed with more threads than cores. Our metric is the time for the first 32 processes to finish. For Kernbench and Ogg Vorbis, we also use the memory-based `tmpfs` file system to reduce performance variability due to disk I/O.

4.2 SMP Evaluation

In this section, we present results that show AMPS improves performance, fairness, and repeatability. We also evaluate the migration overheads that AMPS introduces. For all experiments, we set six cores of our SMP system to run at 50% of their full duty cycles.

Performance. Figure 5 shows our results for each benchmark. Compared to stock Linux, AMPS achieves a median speedup of 1.16 and a maximum of 1.44. Both SPEC OMP* and Kernbench spend most time in parallel phases and a small fraction in sequential phases. In the sequential phases, only one thread runs and AMPS enables it to always run on a fast core. In the parallel phases, whenever a thread finishes on a fast core, AMPS moves another to it from a slow core. Thus, for SPEC OMP* and Kernbench, faster-core-first scheduling enables higher performance. Both SPECjbb2005* and Ogg Vorbis over-subscribe the system with more threads than cores. Since all cores are constantly busy, AMPS does not improve performance noticeably. Nonetheless, we find that AMPS improves fairness and repeatability for these benchmarks.

Fairness. In an over-subscribed system, asymmetry-aware load balancing in AMPS enables threads to share cores fairly. We run Ogg Vorbis and measure the runtimes of the 32 processes that complete first. We repeat this experiment for both stock Linux and Linux extended with AMPS. Figure 6 shows the distribution of the 32 processes' runtimes, i.e., the fraction of processes whose runtime is within a given interval on the x-axis. With stock Linux, 25% of the processes finish in 120 to 130 seconds and 72% between 240 and 250. If each process belongs to a different client, these data indicate that clients who run the same application can receive significantly different performance. Thus, Linux fails to achieve fair sharing of core computing power for clients with equal priority. With AMPS, 88% of the processes have a runtime between 180 and 210 seconds. Different processes now have much smaller differences in performance, indicating that clients share the cores

Table 2: Benchmarks.

SPEC OMP* V3.1: We use the medium versions of the 11 benchmarks with the reference inputs. We compile them using Intel® compilers 9.0 with flags "-fast -openmp" for Fortran programs and "-fast -openmp -ansi_alias" for C. We set the number of OpenMP threads equal to the number of cores in the system.
SPECjbb2005*: We use SPECjbb2005* V1.06 and BEA* JRockit* 5.0 JVM. Each run starts with one warehouse (thread) and continues up to $2 * N$ warehouses, where N is the number of cores in the system.
Kernbench: We use the parallel make benchmark, Kernbench v0.30, to compile the Linux 2.6.15.1 kernel source. We use $4 * N$ make threads, where N is the number of cores in the system.
Ogg Vorbis: We use the Ogg Vorbis audio encoder OggEnc v1.0.1. The encoder is single-threaded. To obtain a multi-threaded workload, we run $4 * N$ copies (processes) of OggEnc, where N is the number of cores in the system, and provide each encoder with its own copy of a 60 MB wav input file.

Table 3: Software overhead results.

Benchmark	Migrations		Overhead
	Linux	AMPS	
Ammp	97	14,146	0.006%
Applu	150	263,846	0.4%
Apsi	63	17,404	0.03%
Art	11	22	0.00002%
Equake	108	124,256	0.4%
Fma3d	139	64,078	0.09%
Gafort	50	25,124	0.02%
Galgel	82	394,817	0.2%
Mgrid	213	317,424	0.2%
Swim	63	59,766	0.06%
Wupwise	73	75,359	0.1%

more fairly with AMPS.

Repeatability. AMPS improves repeatability of application performance across different runs because it schedules threads to faster cores whenever possible. However, with stock Linux, a thread may execute on a faster core in one run but on a slow core in another, causing large variations in application performance between different runs. Figure 7 shows our results for SPECjbb2005*. With stock Linux, throughput drops from four warehouses (i.e., threads) to five. We ran this benchmark three times and observed the same behavior in two runs; in the third run, throughput at five warehouses became higher than that at four. This variability is due to that stock Linux dispatched all of the five SPECjbb2005* threads to slow cores in the first two runs, while, in the third run, it dispatched two threads to the fast cores and the remaining three to the slow cores. In contrast, the results with AMPS are more deterministic and consistent across different runs.

Migration Overhead. We use SPEC OMP* to evaluate the migration overhead that AMPS introduces. For each benchmark, we ran it once and show the profiling results. As we discussed in Section 2.4.1, the migration overhead includes both software and hardware overhead. For software overhead, we instrumented Linux to measure the number of thread migrations and the time each migration takes to complete the four steps in Section 2.4.1. For the entire execution of each benchmark, we calculate its total *increase* in migration time that AMPS introduces over stock Linux. Table 3 shows our results. The overhead column shows the percentage of the extra migration time with AMPS to the total runtime of each benchmark on stock Linux. We see that AMPS introduces a large number of extra migrations; however, they only result in negligible amounts of overhead in terms of runtime. Besides the four steps we measured, a *running* thread's migration has two extra steps: awakening a migration thread and context-switching it and the running thread (Section 3). These steps are very efficient in Linux. Thus, we believe that they do not impact our results.

For the hardware overheads associated with migrations, we use processor performance counters to measure a set of hardware events for both stock Linux and AMPS. For each event, we count its occurrences in the entire execution of each SPEC OMP* benchmark and aggregate per-core counters to obtain whole-system event counts. For each SPEC OMP* benchmark, our results show that miss ratios of the instruction TLB, data TLB, and trace cache with stock Linux and AMPS are nearly identical to each other (all ratios less than 5%). Thus, the extra migrations that AMPS introduces have negligible impact in terms of TLB and trace cache performance.

Figure 8 compares the total L1 data and L2 unified cache misses for SPEC OMP* on stock Linux and AMPS. The value above each AMPS bar shows its percentage increase in cache misses relative to stock Linux. For all benchmarks, the differences in L1 misses between stock Linux and AMPS are at most 4%; for all but `Mgrid`, the L2 differences are at most 6%. Thus, for most benchmarks, the hardware overhead with AMPS is negligible. In repeated measurements, however, `Mgrid` with AMPS shows 13% fewer L2 misses than with stock Linux, which is counter-intuitive since one would expect more L2 misses with AMPS due to its extra migrations. The rest of this section focuses on understanding this result.

Our investigation has revealed that this result is due to hardware prefetching being more effective with AMPS, thus reducing the number of L2 misses. Disabling hardware prefetching and re-running `Mgrid` shows that Linux and AMPS now generate almost the same number of L2 misses. With prefetching enabled, OProfile reveals that 40% of the L2 misses occur in the `resid()` function in `Mgrid`. To isolate the problem, we wrote a microbenchmark to mimic the code in `resid()`. Our microbenchmark creates N threads and assigns each thread an equal-sized portion of a 256 MB global array. The N threads simultaneously read through its portion in fixed strides. Such memory accesses are similar to those in `resid()` and are particularly suitable for hardware prefetching.

We first ran the microbenchmark on stock Linux with all eight fast cores and no hardware prefetching. We varied the number of threads from one to eight and observed that the number of L2 misses stayed almost constant. We then enabled prefetching and re-ran the tests. Figure 9(a) shows our L2 miss results, which indicates that prefetching is less effective as the number of threads increases. Figure 9(b) explains these results by plotting the average latency per prefetch and data read request as a function of the number of threads. As the number of simultaneously running threads increases, the amount of memory traffic increases, caus-

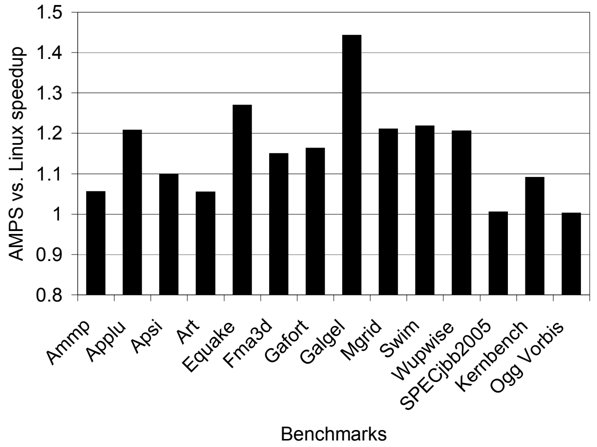


Figure 5: AMPS vs. Linux speedups on an eight-core SMP with two fast and six slow cores.

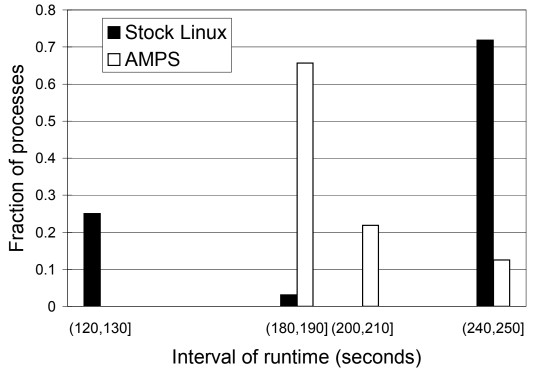
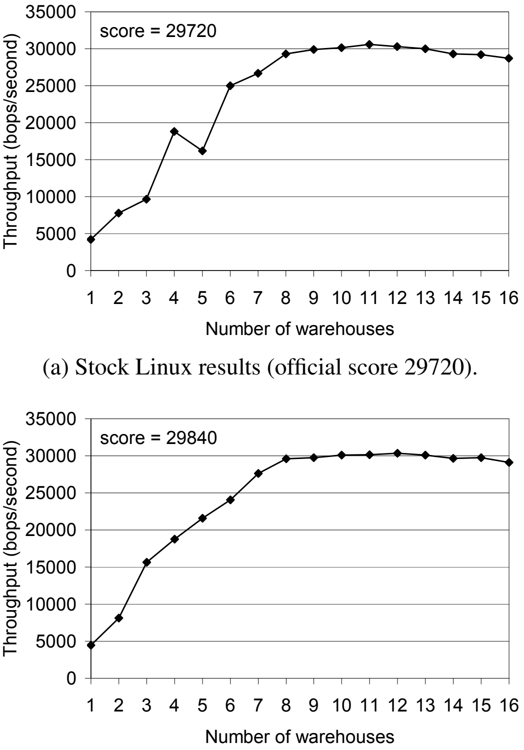


Figure 6: Ogg Vorbis SMP results showing better fairness with AMPS than stock Linux.



(a) Stock Linux results (official score 29720).

(b) AMPS results (official score 29840).

Figure 7: SPECjbb2005* SMP results showing better repeatability with AMPS than stock Linux.

ing each core's internal bus queuing delay to increase and, consequently, prefetch latencies to increase. Therefore, when there are more threads, prefetching becomes less effective and overall L2 misses increase.

On an asymmetric system, both our microbenchmark and `Mgrid` initially have eight threads running. When the two threads on the faster cores finish, AMPS moves two other threads to them. Thus, the number of simultaneously running threads decreases over time from eight to six, four, and finally two. With stock Linux, threads cannot migrate to the faster cores when they are idle. Therefore, for a larger fraction of time, six threads on the slow cores simultaneously access memory. The higher bus traffic causes prefetching to be less effective in stock Linux and, consequently, more L2 misses.

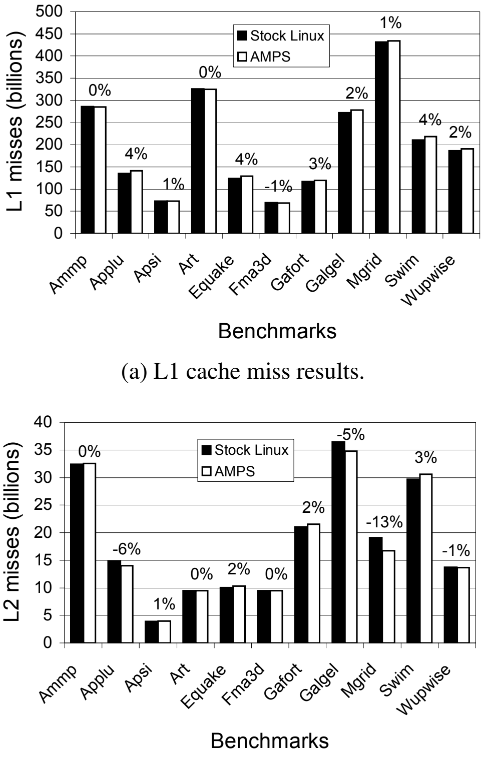
The effects of prefetching can change if the system's asymmetry configuration changes. We varied the number of fast cores from zero (all slow) to eight (all fast) with prefetching enabled and measured L2 misses for both stock Linux and AMPS. Figure 10 shows our results. We see that with two fast cores, AMPS has the minimum number of L2 misses. With four fast cores, prefetching becomes less effective due to increased bus traffic, causing AMPS to have more L2 misses than stock Linux. Beyond four, both designs generate similar amounts of bus traffic and thus incur a similar number of L2 misses.

4.3 NUMA Evaluation

In this section, we evaluate AMPS for two NUMA configurations. In *NUMA-1*, we keep the eight cores in the first two nodes of our system at their full duty cycles and set the remaining 24 cores to run at 50% of their full duty cycles. In *NUMA-2*, we spread the faster cores such that each of the eight nodes contains one faster core. These configurations represent two design points in future architectures: one that has system-wide asymmetry and one that constrains asymmetry within a socket. For both configurations, we run SPEC OMP* and evaluate the Always, Same-Node, and RSS policies described in Section 2.4.3. Figure 11 shows our results.

For NUMA-1, the Always policy performs much worse than stock Linux for most benchmarks due to significant migration overheads. However, `Ammp` and `Galgel` obtain good speedups because they have small per-thread working sets and frequent migrations have little impact. Surprisingly, `Galgel` running on stock Linux causes our OpenMP runtime to trigger a rare Linux problem that leads to heavy contention on a kernel spinlock [17]. With AMPS, the contention is much lower because often fewer threads are running simultaneously, as we discussed in the cache analysis of `Mgrid`. The Same-Node policy brings performance close to stock Linux, as it constrains migrations within the same node. However, since the faster cores concentrate only in the first two nodes, threads on other nodes cannot benefit from them. The RSS policy solves this problem and leads AMPS to outperform stock Linux for every benchmark with speedups ranging from 1.02 to 2.61.

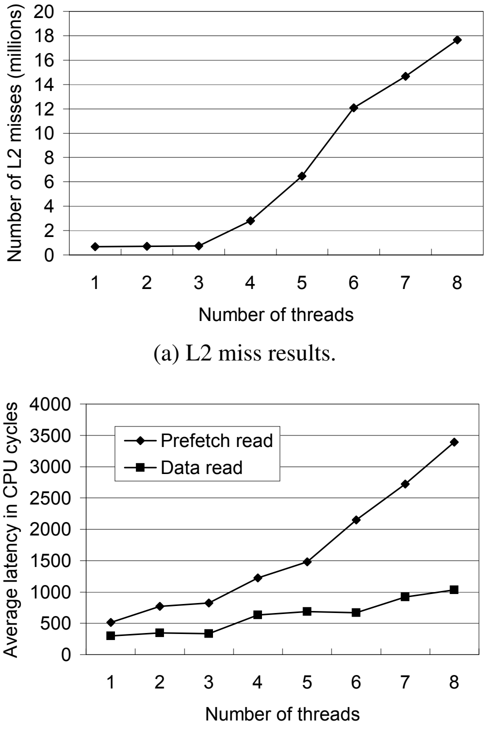
For NUMA-2, the Always and Same-Node policies perform much better than in NUMA-1 because many migrations to faster cores occur within the same node. For these migrations, cache misses are often resolved in local memory, especially since our system maintains a per-node L4 cache. Among the three policies, the RSS



(a) L1 cache miss results.

(b) L2 cache miss results.

Figure 8: SPEC OMP* cache miss results on the SMP system. Above each AMPS bar is its percentage increase in cache misses relative to stock Linux.



(a) L2 miss results.

(b) Prefetch and data latency.

Figure 9: Microbenchmark results for stock Linux.

policy performs the best with speedups ranging from 1.02 to 1.49 over stock Linux. One exception is `Applu`, whose performance degrades slightly compared to stock Linux, which we believe is due to the inaccuracies in our overhead prediction algorithm, as discussed in Section 2.4.4.

For NUMA-1 and NUMA-2, the RSS policy achieves an overall median speedup of 1.07 and 1.18 on average over stock Linux. For some benchmarks, the NUMA speedups are lower than the SMP ones because, on the SMP, AMPS enables threads to migrate to faster cores whenever they are under-utilized, thus fully exploiting their performance advantages. However, their advantages on NUMA are not always fully exploited because threads sometimes cannot migrate to a faster core due to high migration overhead.

To see how the migration policies affect thread migrations, we measure the number of migrations in each benchmark with stock Linux and AMPS using each of the three policies. Due to space constraints, we show only the results for NUMA-1 in Table 4. For NUMA-2, our results show that each benchmark has more migrations and the gaps between the Always and Same-Node policies are much smaller. From Table 4, we see that the RSS policy constrains migrations for benchmarks with large working sets, but allows more migrations for `Ammp` and `Galgel`, whose working sets are relatively small. Such adaptability enables the RSS policy to achieve the best performance among the three policies.

5. RELATED WORK

Research on conventional multiprocessors [1, 18] showed that performance-asymmetric (or heterogeneous) architectures achieve higher performance than cost-equivalent homogeneous ones. To efficiently schedule tasks, prior research [4, 21, 23] used graph modeling of heterogeneous hardware and task properties. However, these algorithms have limited use in practice due to simplified and often unrealistic assumptions, such as *a priori* knowledge of task runtime and dependencies. Figueiredo and Fortes [10] studied heterogeneous distributed shared-memory (DSM) multiprocessors and proposed a static scheduling policy that uses processor performance ratios. This policy is similar to our asymmetry-aware load balancing algorithm, but different in that it performs only static assignments, whereas our algorithm performs dynamic load balancing. Bender and Rabin [5] proposed a high utilization scheduler for heterogeneous processors. They proved that this scheduler is almost optimal for typical parallel programs and has bounded migration overhead. This design is similar to our faster-core-first algorithm. Our work differs in that we apply the algorithm to the OS scheduler, as opposed to a language runtime.

With the advances in VLSI technology, recent research advocates heterogeneous multi-core architectures. Kumar et al. [13, 15] proposed sampling-based OS scheduling. DeVuyst et al. [8] studied sampling and electron policies that adapt to thread execution phases. Different from us, all of this research used simulation and did not study tradeoffs of an actual OS implementation. Prior research [2, 3, 11] also used clock modulation to emulate asymmetric architectures. Balakrishnan et al. [3] used a scheduling policy similar to our faster-core-first algorithm and obtained similar results on performance repeatability. Our work differs in that we seek to improve performance and fairness for both SMP and NUMA architectures. Ghiasi et al. [11] described a scheduling algorithm that uses performance counters to predict thread performance at a given core frequency. Their algorithm attempts to reduce power consumption with minimum performance loss and incurs higher overhead than ours. Annavaram et al. [2] studied how to maximize

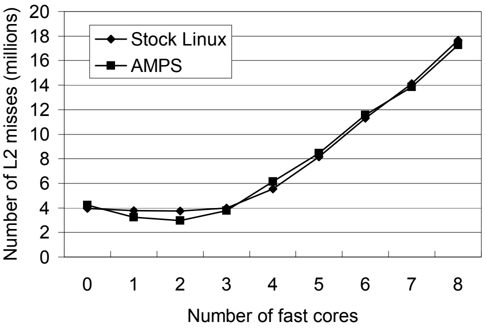


Figure 10: Effects of asymmetry on L2 misses.

Table 4: NUMA-1 number of migrations.

Benchmark	Stock Linux	AMPS		
		Always	Same-Node	RSS
Ammp	62	8,947	180	7,968
Applu	116	95,942	417	328
Apsi	211	47,177	433	261
Art	27	86	25	28
Equake	66	174,265	5,716	206
Fma3d	272	91,520	697	522
Gafort	179	46,621	443	361
Galgel	108	80,178	10,246	36,944
Mgrid	94	457,378	1,307	503
Swim	217	79,278	1,783	1,226
Wupwise	238	102,744	482	376

performance with a fixed power budget. Their scheduling policy assumes hardware support for detecting application sequential and parallel phases, while AMPS makes no such assumption.

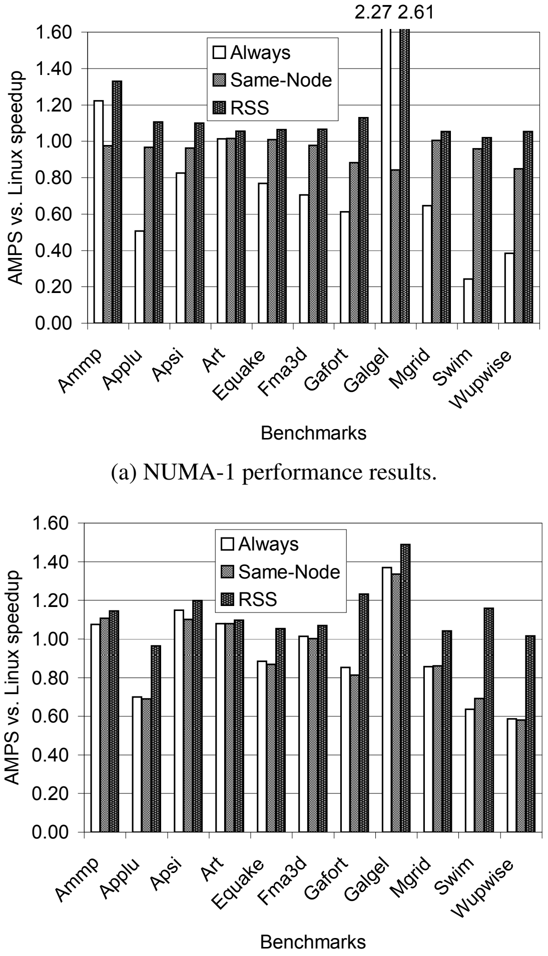
The CELL* processor [19] is an extreme form of asymmetric architecture where cores of different functionalities and ISAs coexist. Hankins et al. [12] proposed ISA extensions that enable applications to explicitly manage functionally asymmetric cores. Uhlig et al. [24] also studied asymmetry-aware scheduling for virtual machines. In contrast, we focus on performance-asymmetric platforms and OS scheduling.

There is a large body of work on NUMA scheduling and memory management (see [7, 16] for detailed treatment of related work). Most of previous work studied dynamic page migration, whereas we focus on thread migration. Existing OSes, such as Linux, consider a thread cache-hot if the time since its last execution on the local processor is less than some threshold and migrate it only if non-cache-hot. Our RSS policy takes into account threads' memory usage and thus more accurately predicts migration overhead.

6. CONCLUSION

As the industry moves toward large-scale multi-core processors, it is increasingly important to investigate architectures that scale up both single-threaded performance and multithreaded throughput. Performance-asymmetric architectures provide a cost-effective solution. However, traditional OSes assume homogeneous hardware and cannot efficiently manage hardware in an asymmetric architecture. To bridge this gap, we proposed the AMPS OS scheduler that efficiently manages both SMP- and NUMA-style performance-asymmetric architectures.

AMPS contains three components: (1) asymmetry-aware load balancing, which balances threads to cores proportionately to their computing power, (2) faster-core-first scheduling, which schedules threads to faster cores whenever they are under-utilized, and (3) NUMA-aware migration, which controls thread migrations based on predictions of their overheads. These components complement one another, collectively providing efficient support for asymmetric architectures. AMPS is easy to deploy as it requires simple modifications to existing OSes and no changes in applications. Our evaluation demonstrated that AMPS improves stock Linux for asymmetric systems in three aspects: performance, fairness, and repeatability of performance measurements.



(a) NUMA-1 performance results.

(b) NUMA-2 performance results.

Figure 11: SPEC OMP* NUMA results.

Our future research will explore the scheduler design space. We plan to extend AMPS to optimize for both performance and power consumption, and study thread-dependent scheduling policies that dynamically monitor and adapt to application runtime behavior.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and suggestions on the early draft of this paper. We thank Jesse Barnes, Jim Held, Barbara Hohlt, Suresh Siddha, and Jessica Young for helpful discussions of this work.

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 *Proceedings 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 *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, 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] M. A. Bender and M. O. Rabin. Scheduling Cilk multithreaded parallel programs on processors of different speeds. In *Proceedings of the Twelfth ACM Symposium on Parallel Algorithms and Architectures*, pages 13–21, July 2000.

[6] S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski, and J. Rattner. Platform 2015: Intel® processor and platform evolution for the next decade. White Paper, Intel Corporation, 2005.

[7] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–24, Oct. 1994.

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

[9] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 395–398, Apr. 2005.

[10] R. J. O. Figueiredo and J. A. B. Fortes. Impact of heterogeneity on DSM performance. In *Proceedings of the Sixth IEEE 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*, pages 199–210, May 2005.

[12] 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 *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 114–127, June 2006.

[13] 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 *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[14] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessing. *IEEE Computer*, 38(11):32–38, Nov. 2005.

[15] 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 *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 64–75, June 2004.

[16] R. P. LaRowe, Jr., C. S. Ellis, and L. S. Kaplan. The robustness of NUMA memory management. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 137–151, Oct. 1991.

[17] Linux Kernel Mailing List. Scalability of signal delivery for POSIX threads. http://lkml.org/lkml/2004/11/22/432, Nov. 2004.

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

[19] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first generation CELL* processor. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 184–185, Feb. 2005.

[20] M. D. Powell, M. Gomaa, and T. Vijaykumar. Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 260–270, Oct. 2004.

[21] 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.

[22] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Nov. 2000.

[23] 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.

[24] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, May 2004.

[25] H. Zheng and J. Nieh. SWAP: A scheduler with automatic process dependency detection. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, pages 183–196, Mar. 2004.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.