Operating System Support for Mitigating Software Scalability Bottlenecks on Asymmetric Multicore Processors

Juan Carlos Saez
Complutense University,
Madrid, Spain
jcsaezal@fdi.ucm.es

Alexandra Fedorova
Simon Fraser University,
Vancouver BC, Canada
fedorova@cs.sfu.ca

Manuel Prieto
Complutense University,
Madrid, Spain
mpmatias@dacya.ucm.es

Hugo Vegas
Complutense University,
Madrid, Spain
hugovegas@fdi.ucm.es

ABSTRACT

Asymmetric multicore processors (AMP) promise higher performance per watt than their symmetric counterparts, and it is likely that future processors will integrate a few *fast* out-of-order cores, coupled with a large number of simpler, *slow* cores, all exposing the same instruction-set architecture (ISA). It is well known that one of the most effective ways to leverage the effectiveness of these systems is to use fast cores to accelerate sequential phases of parallel applications, and to use slow cores for running parallel phases. At the same time, we are not aware of any implementation of this *parallelism-aware* (PA) scheduling policy in an operating system. So the questions as to whether this policy can be delivered efficiently by the *operating system* to unmodified applications, and what the associated overheads are remain open. To answer these questions we created two different implementations of the PA policy in OpenSolaris and evaluated it on real hardware, where asymmetry was emulated via CPU frequency scaling. This paper reports our findings with regard to benefits and drawbacks of this scheduling policy.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling

General Terms

Performance, Measurement, Algorithms

Keywords

Asymmetric multicore, Scheduling, Operating Systems

1. INTRODUCTION

An asymmetric multicore processor (AMP) includes cores exposing the same instruction-set architecture, but differing in features, size, speed and power consumption [2, 8]. A typical AMP would contain a number of simple, small and low-power *slow* cores and a few complex, large and high-power *fast* cores. It is well known that AMP systems can mitigate scalability bottlenecks in parallel applications by accelerating sequential phases on fast cores [2, 7, 12].

To leverage this potential of AMP systems, threads must be mapped to cores in consideration of the amount of parallelism in the application: if an application is highly parallel its threads should be mapped to slow cores, but if the application is sequential or is executing a sequential phase its thread should be mapped to a fast core. A natural place for this *Parallelism-Aware* (PA) policy is in the operating system. This way, many applications can reap its benefits, potentially without requiring any modifications, and the sharing of scarce fast cores among multiple applications can be fairly arbitrated by the operating system. To the best of our knowledge, there are no OS-level implementations of the PA scheduling policy. As a result, many questions regarding the effectiveness and practicality of this policy remain open.

One open question is *how can the operating system effectively detect sequential phases in applications?* In some applications unused threads block during the sequential phase, and by monitoring the application's runnable thread count, which is exposed to the OS by most threading libraries, the scheduler can trivially detect a sequential phase. In other applications, however, unused threads busy-wait (or spin) during short periods of time, and so the OS cannot detect these phases simply by monitoring the runnable thread count. To address these scenarios we designed PA Runtime Extensions (PA-RTX) – an interface and library enhancements enabling the threading library to notify the scheduler when a thread spins rather than doing useful work. We implemented PA-RTX in a popular OpenMP runtime, which required only minimal modifications to support them, but the extensions are general enough to be used with other threading libraries.

Another open question is the overhead associated with the PA policy. Any policy that prioritizes fast cores to specific threads is bound to generate migration overheads – a performance degradation that occurs when a thread is moved from one core to another. Performance degradation results from the loss of cache state accumulated on the thread's old core. Upon evaluating these overheads we found that they can be significant (up to 18%) if the fast core is placed in a different memory hierarchy domain from slow cores, but a hardware configuration where a fast core shares a memory hierarchy domain with several slow cores coupled with a topology-aware scheduler practically eliminates these overheads.

We evaluate the PA policy on a real multicore system, where "slow" cores were emulated by reducing the clock frequency on the processors, while "fast" cores were configured to run at regular speed. We find that the main benefits from the PA policy are derived for multi-application workloads and when the number of fast cores relative to slow cores is small. In this case, it delivers speedups of up to 40% relative to the OpenSolaris default asymmetry-agnostic scheduler. Previously proposed asymmetry-aware algorithms, which we used for comparison, also do well in some cases, but unlike our parallelism-aware algorithms they do not perform well across the board, because they fail to consider the parallelism of the application.

The key contribution of our work is the evaluation of the operating system technology enabling next-generation asymmetric systems. We are not aware of previous studies investigating the benefits and drawbacks of the PA scheduling policy implemented in a real OS. Our findings provide insights for design of future asymmetry-aware operating systems and asymmetric hardware alike.

The rest of the paper is structured as follows. Section 2 presents the design and implementation of the PA scheduling algorithm. Section 3 presents experimental results. Section 4 discusses related work. Section 5 summarizes our findings and discusses future work.

2. DESIGN AND IMPLEMENTATION

In Section 2.1 we describe two parallelism-aware algorithms proposed in this work: PA and MinTLP. In Section 2.2 we describe the runtime extensions to PA (PA-RTX). A brief description of other asymmetry-aware algorithms that we use for comparison is provided in Section 2.3.

2.1 PA and MinTLP algorithms

Our algorithms assume an AMP system with two core types: *fast* and *slow*. Previous studies concluded that supporting only two core types is optimal for achieving most of the potential gains on AMP [8]; so we expect this configuration to be typical of future systems. More core types may be present in future systems due to variations in the fabrication process. In that case, scheduling must be complemented with other algorithms, designed specifically to address this problem [18].

The goal of the algorithm is to decide which threads should run on fast cores and which on slow cores. In MinTLP, this decision is straightforward: the algorithm selects applications with the smallest thread-level parallelism (hence the name MinTLP) and maps threads of these applications to fast cores. Thread-level parallelism is determined by examining the number of runnable (i.e., not blocked) threads. If not enough fast cores are available to accommodate all these threads, some will be left running on slow cores. MinTLP makes no effort to fairly share fast cores among all "eligible" threads. This algorithm is very simple, but not always fair.

The other proposed algorithm, PA, is more sophisticated. It classifies threads dynamically into several categories: MP, HP, and SP. The MP (mildly parallel) category includes threads belonging to applications with a low degree of thread-level parallelism, including the single-threaded applications. The HP category includes threads belonging to highly parallel applications. The MP threads will run primarily on fast cores, and the HP threads will run primarily on slow cores. Threads of applications whose runnable thread count exceeds `hp_threshold` fall into the HP category, the remaining threads fall into the MP category.

A special class SP is reserved for threads of parallel applications that have just entered a sequential phase. These threads will get the highest priority for running on fast cores: this provides more opportunities to accelerate sequential phases. To avoid monopolizing fast cores, SP threads are downgraded by the scheduler into the MP class after spending `amp_boost_ticks` scheduling clock ticks in the SP class.

If there are not enough cores to run all SP and MP threads on fast cores, the scheduler will run some of the threads on slow cores, to preserve load balance. SP threads have a higher priority in using fast cores. The remaining fast cores will be shared among MP threads in a round-robin fashion.

The scheduler keeps track of the count of runnable threads in each applicationto detect transitions between the aforementioned classes and perform thread-to-core mapping adjustments accordingly. To avoid premature migrations and preserve load balance, PA integrates a thread swapping mechanism to perform those adjustments periodically, instead of reacting to those transitions immediately (MinTLP also integrates a similar swapping mechanism).

When the change in the thread-level parallelism cannot be determined via the monitoring of the runnable thread count, PA relies on the Runtime Extensions, described in the next Section. We must also highlight that despite the fact that our evaluation has been focused on multi-threaded single-process applications, the PA and MinTLP algorithms can be easily extended to support multi-process applications using high-level abstractions provided by the operating system, such as process sets.

Although sensitivity of the PA algorithm to its configurable parameters was studied, we are unable to provide the results due to space constraints. We found, however, that it is generally easy to choose good values for these parameters. After performing such a sensitivity study, we set `amp_boost_ticks` to one hundred timeslices (1 second) and `hp_threshold` to one greater than the number of fast cores. These values ensure acceleration of sequential phases without monopolizing fast cores.

2.2 PA Runtime Extensions

The base PA algorithm introduced so far relies on monitoring runnable thread count to detect transitions between serial and parallel phases in the application. However, conventional synchronization primitives found in most threading libraries use an adaptive two-phase approach where unused threads busy wait for a while before blocking to reduce context-switching overheads. While blocking is coordinated with the OS, making it possible to detect phase transitions, spinning is not. Reducing the spinning phase enables the OS to detect more serial phases. However, in our context it may also lead to excessive migrations and cause substantial overheads (as soon as a fast core becomes idle PA and MinTLP will immediately migrate a thread to this core). In the event these busy-waiting phases are frequent, it is helpful to give the scheduler some hints that would help it to avoid mapping spinning threads to fast cores. To that end, we propose two optimizations, which can be implemented in the threading library (applications themselves need not be changed).

2.2.1 *Spin-then-notify mode*

Our first proposal is a new *spin-then-notify* waiting mode for synchronization primitives. Its primary goal is to avoid running spinning threads on fast cores and save these "power-hungry" cores for other threads. In this mode the synchronization primitive notifies the operating system via a system call after a certain *spin threshold* that the thread is busy-waiting rather than doing useful work. Upon notification, the PA scheduler marks this thread as a *candidate* for migration to slow cores. We have opted to mark threads as migration candidates instead of forcing an immediate migration since this approach avoids premature migrations and allows a seamless integration with the PA and MinTLP swapping mechanisms. The synchronization primitive also notifies the scheduler when a spinning thread finishes the busy wait. In Section 3.2 we explore the advantages of using the new spin-then-notify mode. For this purpose we have modified the OpenMP runtime system to include this new mode in the basic waiting function used by high-level primitives such as mutexes or barriers.

Another potentially useful feature of this primitive may arise in the context of scheduling algorithms that map threads on AMP systems based on their relative speedup on fast vs. slow cores (see Section 4). These algorithms typically measure performance of each thread on fast and slow cores and compute its performance ratio, which determines the relative speedup [5, 9]. If a thread performs busy-waiting it can achieve a very high performance ratio, since a spin loop uses the CPU pipeline very efficiently[1]. As a result, the proposed algorithms would map spinning threads to fast cores despite they are not doing useful work. Even though these implementation issues could be solved via additional hardware support [11], a spin-then-notify primitive could help avoid the problem without needing extra hardware.

2.2.2 *Exposing the master thread*

We have also investigated a simple but effective optimization allowing the application to communicate to the kernel that a particular thread must have a higher priority in running on a fast core. This optimization was inspired by the typical structure of OpenMP *do-all* applications. In these applications, there is usually a *master thread* that is in charge of the explicit serial phases at the beginning, in between parallel loops, and at the end of the application (apart from being in charge of its share of the parallel loops). Identifying this master thread to the kernel enables the scheduler to give it a higher priority on the fast core simply because this thread will likely act as the "serial" thread. This hint can speed up do-all applications even without properly detecting serial phases. Our PA Runtime Extensions enable the runtime system to identify the master thread to the scheduler via a new system call. If the pattern of the application changes and another thread gets this responsibility, the same system call can be used to update this information.

To evaluate this feature, we have modified the OpenMP runtime system to automatically identify the thread executing the *main* function as the master thread to the kernel, right after initializing the runtime environment. In the same way as the implementation of spin-notify mode, only the OpenMP library needs to be modified, not requiring any change in the applications themselves. Upon receiving this notification, the PA scheduler tries to ensure that the master thread runs on a fast core whenever it is active, but without permanently binding the thread to that core as would be done with other explicit mechanisms based on thread affinities. This way, PA still allows different threads to compete for fast cores according to its policies.

2.3 The other schedulers

We compare PA and MinTLP to three other schedulers proposed in previous work. Round-Robin (RR) equally shares fast and slow cores among all threads [5]. BusyFCs is a simple asymmetry-aware scheduler that guarantees that fast cores never go idle before slow cores [4]. Static-IPC-Driven, which we describe in detail below, assigns fast cores to those threads that experience the greatest relative speedup (in terms of instructions per second) relative to running on slow cores [5]. We implemented all these algorithms in OpenSolaris. Our baseline for comparison is the asymmetry-agnostic default scheduler in OpenSolaris, referred to hereafter as Default.

The Static-IPC-Driven scheduler is based on the design proposed by Becchi and Crowley [5]. Thread-to-core assignments in that algorithm are done based on per-thread IPC ratios (quotients of IPCs on fast and slow cores), which determine the relative benefit of running a thread on a particular core type. Threads with the highest IPC ratios are scheduled on fast cores while remaining threads are scheduled on slow cores. In the original work [5], the IPC-driven scheduler was simulated. This scheduler samples threads' IPC on cores of all types whenever a new program phase is detected. Researchers who attempted an implementation of this algorithm found that such sampling caused large overheads, because frequent cross-core thread migrations were required [16]. To avoid these overheads, we have implemented a static version of the IPC-driven algorithm, where IPC ratios of all threads are measured *a priori*. This makes IPC ratios more accurate in some cases [16] and eliminates much of the runtime performance overhead. Therefore, the results of the Static-IPC-Driven scheduler are somewhat optimistic and the speedups of PA and MinTLP relative to Static-IPC-Driven are somewhat pessimistic.

3. EXPERIMENTS

The evaluation of the PA algorithm was performed on an AMD Opteron system with four quad-core (Barcelona) CPUs. The total number of cores was 16. Each core has private 64KB instruction and data caches, and a private L2 cache of 512KB. A 2MB L3 cache is shared by the four cores on a chip. The system has a NUMA architecture. Access to a local memory bank incurs a shorter latency than access to a remote memory bank. Each core is capable of running at a range of frequencies from 1.15 GHz to 2.3 GHz. Since

[1]Best practices in implementing spinlocks dictate using algorithms where a thread spins on a local variable [1], which leads to a high instruction throughput.

Table 1: Classification of selected applications.

Categories	Benchmarks
HP-CI	`EP`(N), `vips`(P), `fma3d`(O), `ammp`(O), `RNA`(I), `scalparc`(M), `wupwise` (O)
HP-MI	`art`(O), `equake`(O), `applu`(O), `swim`(O)
PS-CI	`BLAST`(NS), `swaptions`(P), `bodytrack`(P), `semphy`(M), `FT`(N)
PS-MI	`MG`(N), `TPC-C`(NS), `FFTW`(NS)
ST-CI	`gromacs`(C), `sjeng`(C), `gamess`(C), `gobmk`(C), `h264ref`(C), `hmmer`(C), `namd`(C)
ST-MI	`astar`(C), `omnetpp`(C), `soplex`(C), `milc`(C), `mcf`(C), `libquantum`(C)

Table 2: Multi-application workloads, Set #1.

Workload name	Benchmarks
STCI-PSMI	`gamess`, `FFTW` (12,15)
STCI-PSCI	`gamess`, `BLAST` (12,15)
STCI-PSCI(2)	`hmmer`, `BLAST` (12,15)
STCI-HP	`gamess`, `wupwise` (12,15)
STCI-HP(2)	`gobmk`, `EP` (12,15)
STMI-PSMI	`mcf`, `FFTW` (12,15)
STMI-PSCI	`mcf`, `BLAST` (12,15)
STMI-HP	`astar`, `EP` (12,15)
PSMB-PSCI	`FFTW` (6,8), `BLAST` (7,8)
PSMB-HP	`FFTW` (6,8), `wupwise_m` (7,8)
PSCI-HP	`BLAST` (6,8), `wupwise_m` (7,8)
PSCI-HP(2)	`semphy` (6,8), `EP` (7,8)

each core is within its own voltage/frequency domain, we are able to vary the frequency for each core independently. We experimented with asymmetric configurations that use two core types: "fast" (a core set to run at 2.3 GHz) and "slow" (a core set to run at 1.15 GHz). We also varied the number of cores in the experimental configurations by disabling some of the cores.

We used three AMP configurations in our experiments: (1) 1FC-12SC – one fast core and 12 slow cores, the fast core is on its own chip and the other cores on that chip are disabled; (2) 4FC-12SC – four fast cores and 12 slow cores, each fast core is on a chip with three slow cores; (3) 1FC-3SC – one fast core, three slow cores, all on one chip. Not all configurations are used in all experiments.

Although thread migrations can be effectively exploited by asymmetry-aware schedulers (e.g. to map sequential parts of parallel applications on fast cores), the overhead that they may introduce can lead to performance degradation. Since we also aim to assess the impact of migrations on performance we opted to select the default asymmetry-unaware scheduler used in OpenSolaris (we refer to it as Default henceforth) as our baseline scheduler. Despite Default keeps threads on the same core for most of the execution time and thus minimizes thread migrations, its asymmetry-unawareness leads it to offer much more unstable results from run to run than the ones observed for the other schedulers. For that reason, a high number of samples were collected for this scheduler, in an attempt to capture the average behavior more accurately. Overall, we found that Default usually fails to schedule single-threaded applications and sequential phases of parallel application on fast cores, especially when the number of fast cores is much smaller than the number of slow cores, such as on the 1FC-12SC and 4FC-12SC configurations.

We evaluate the base implementation of the PA algorithm as well PA with Runtime Extensions. We compare PA to RR, BusyFCs, Static-IPC-Driven, Min-TLP and to Default. In all experiments, each application was run a minimum of three times, and we measure the average completion time. The observed variance was small in most cases (so it is not reported) and where it was large we repeated the experiments for a larger number of trials until the variance reached a low threshold. In multi-application workloads the applications are started simultaneously and when an application terminates it is restarted repeatedly until the longest application in the set completes at least three times. We report performance as the speedup over Default. The geometric mean of completion times of all executions for a benchmark under a particular asymmetry-aware scheduler is compared to that under Default, and percentage speedup is reported.

In all experiments, the total number of threads (sum of the number of threads of all applications) was set to match the number of cores in the experimental system, since this is how runtime systems typically configure the number of threads for the CPU-bound workloads that we considered [19].

Our evaluation section is divided into four parts. In Section 3.1 we introduce the applications and workloads used for evaluation. In Section 3.2 we evaluate PA runtime extensions. In Section 3.3 we evaluate multi-application workloads. Finally, in Section 3.4 we study the overhead.

3.1 Workload selection

We used applications from PARSEC [6], SPEC OMP2001, NAS [3] Parallel Benchmarks and MineBench [13] benchmark suites, as well as the TPC-C benchmark implemented over Oracle Berkeley DB [14], BLAST – a bioinformatics benchmark, FFT-W – a scientific benchmark performing the fast Fourier transform, and RNA – an RNA sequencing

Table 3: Multi-application workloads, Set #2.

Workload name	Benchmarks
2STCI-2STMI-1HP	`gamess`, `h264ref`, `astar` , `soplex`, `wupwise` (12)
4STCI-1HP	`gromacs`, `gamess`, `namd`, `gobmk`, `EP` (12)
3STCI-1STMI-1PSCI	`gamess`, `hmmer`, `gobmk`, `soplex`, `semphy` (12)
2STCI-1STMI-1PSMI-1HP	`gamess`, `h264ref`, `soplex`, `FFTW` (6), `equake` (7)
3STCI-3STMI-1HP	`gromacs`, `sjeng`, `h264ref`, `libquantum`, `milc`, `omnetpp`, `EP` (10)
3STCI-3STMI-1PSCI	`gromacs`, `sjeng`, `h264ref`, `libquantum`, `milc`, `omnetpp`, `BLAST` (10)

application. For multi-application workloads we also used sequential applications from SPEC CPU2006.

We classified applications according to their architectural properties: memory-intensive (MI) or compute-intensive (CI), as well as according to their parallelism: highly parallel (HP), partially sequential (PS) and single-threaded (ST). Memory-intensity was important for fair comparison with Static-IPC-Driven. CI applications have a higher relative speedup on fast cores [16] and so it was important to include applications of both types in the experiments. Parallelism class was determined by tracing execution via OpenSolaris' DTrace framework and measuring the fraction of time the application spent running with a single runnable thread. Parallel applications where this fraction was greater than 7% were classified as PS, whereas the rest were classified as HP. The ST class includes sequential applications. Table 1 shows the classification of our selected applications according to these classes. The text in parentheses next to the benchmark name indicates the corresponding benchmark suite: O –SPEC OMP2001, P– PARSEC, M – Minebench, N– NAS, C – SPEC CPU2006, and NS – other benchmarks not belonging to any specific suite.

By default, all OpenMP applications were compiled with the native Sun Studio compiler. In order to evaluate PA Runtime Extensions (Section 3.2) we had to modify the OpenMP runtime system but the source code for the Sun Studio OpenMP runtime system was not available to us. For that reason, we resorted to using the Linux version of the GCC 4.4 OpenMP runtime system in OpenSolaris[2]. Nevertheless, we observed that the performance of OpenMP applications with Sun Studio and GCC is similar.

Both OpenMP and POSIX threaded applications used in section 3.3 and 3.4 run with adaptive synchronization modes; as such sequential phases are exposed to the operating system in both cases. In these sections we do not use runtime extensions with parallelism-aware algorithms. All OpenMP applications run with the default adaptive synchronization mode used by GCC 4.4 unless otherwise noted (Sun Studio can be easily configured to use a similar adaptive mode). POSIX threaded applications (such as `BLAST` or `bodytrack`) use full blocking modes on all synchronization primitives but on those related to POSIX standard mutexes and synchronization barriers, where an adaptive implementation is provided by OpenSolaris. Unlike OpenMP applications, threads of POSIX applications spin for shorter periods of time before blocking on those adaptive synchronization primitives (these are the default parameters used in OpenSolaris).

[2]Using such a version of the runtime system required augmenting OpenSolaris with a Linux compatible `sys_futex` syscall)

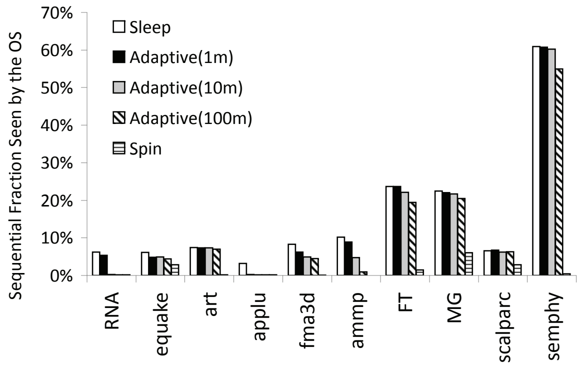


Figure 2: Variations in the sequential fraction seen by the OS when varying the synchronization mode and *blocking threshold*.

For Section 3.2 we selected ten OpenMP applications: `art`, `applu`, `fma3d`, `ammp`, `FT`, `MG`, `scalparc`, `semphy` and `RNA`. These applications were chosen to cover a wide variety of sequential portions. In the overhead section we analyze ten parallel applications accross the aforementioned classes: three HPCI (`RNA`, `wupwise` and `vips`), two HPMI (`swim` and `applu`), three PSCI (`swaptions`, `bodytrack` and `BLAST`) and two PSMI applications (`TPC-C` and `FFTW`) .

For Section 3.3, we constructed two sets of multi-application workloads. The first set, shown in Table 2, comprises twelve representative pairs of benchmarks across the previous categories mentioned above. For the sake of completeness, we experimented with additional multi-application workloads with more than two applications. Table 3 shows this second set, consisting of six workloads.

3.2 PA Runtime Extensions

We begin by investigating the effect on performance when using different synchronization waiting modes under the PA scheduler. In these experiments we demonstrate that using a low blocking threshold effectively exposes sequential phases to the scheduler, but performance can also suffer if the threshold is set too low. Then we evaluate PA-RTX and show that it offers comparable performance to purely adaptive approaches and in some cases even improves it.

In the following experiment we used the 1FC-12SC configuration and tested three different waiting modes: spin, sleep and adaptive. In spin mode unused threads busy-wait for the entire time, in sleep mode, they block immediately. We studied the effects of various synchronization modes on all asymmetry-aware schedulers, but since our results showed

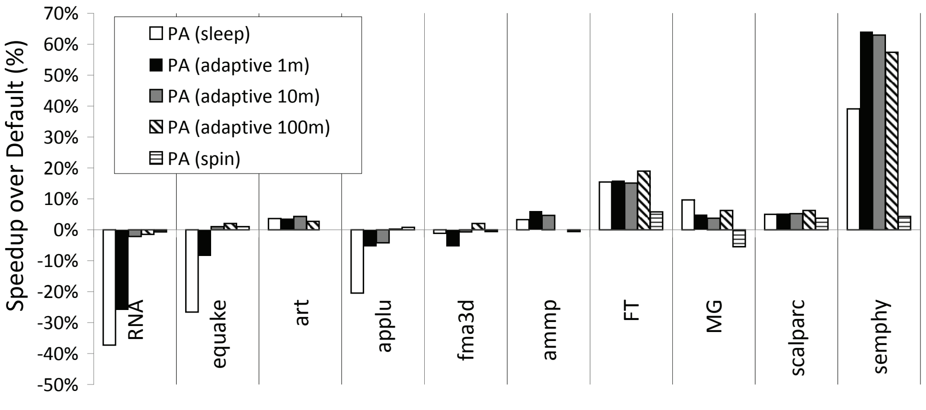


Figure 1: Speedup from PA using sleep, spin and adaptive modes with different *blocking thresholds*.

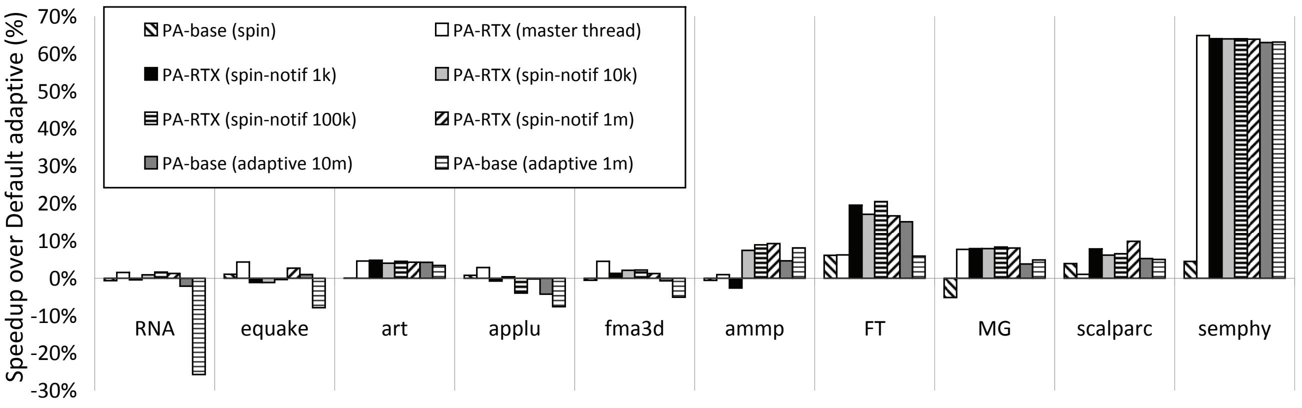


Figure 3: Speedup from PA with Runtime Extensions.

that across the schedulers the effects are largely the same, we present the data for the PA scheduler only.

Figure 1 shows the results. PA runtime extensions are *not* used in this case. When the spin mode is used, the base PA algorithm delivers hardly any speedup, because it is not aware of the sequential phases. With the adaptive and sleep modes, applications on the right side of the chart experience noticeable speedup. They have large sequential phases and switching to the adaptive or sleep mode exposes these phases to the scheduler and enables their acceleration on the fast core. Applications on the left side of the chart, however, experience performance *degradation*. Those with the highest overhead (RNA, `equake` and `applu`) run frequent short parallel loops. Despite being well-balanced applications, the asymmetry of the platform causes the thread running on a fast core to complete its share of these loops earlier. In this case, the sleep mode makes the fast core become idle very often, triggering frequent migrations that introduce substantial overheads. An adaptive mode alleviates this issue, but the blocking threshold must be sufficiently large to remove the overheads completely.

Figure 2 shows how the fraction of time spent in sequential phases as seen by the OS changes for different blocking thresholds. This further underscores that the blocking threshold for the adaptive mode must be chosen carefully: choosing a very large threshold reduces the visibility of sequential phases for the OS, but a small one causes overhead, as shown in Figure 1.

We now evaluate the PA algorithm with Runtime Extensions (PA-RTX). We test the *spin-then-notify* synchronization mode using several spin thresholds. The blocking threshold is set at 100m iterations in all experiments. We also test the feature permitting the application to expose the "master thread". These scenarios are compared with the Default scheduler where applications use the adaptive mode, and with the base PA algorithm (no RTX) using the spin mode (PA-base (spin)) and the adaptive mode (PA-base (adaptive)). For PA-base (adaptive) we use the best blocking thresholds: 10m and 1m respectively. Figure 3 shows the results.

Overall, we conclude that PA-RTX is less sensitive to the choice of thresholds than PA-base (adaptive). PA-base (adaptive) hurts performance for several applications, up to a maximum of as much as 26%(!) when a low value of the blocking threshold is used. PA-RTX hurts performance by 4% at most and only in one case, and that happens when the spin threshold is set to an extremely small value of 1K iterations. With adaptive synchronization, a trade-off must be made when setting the blocking threshold: choosing a small value may hurt performance, but choosing a value that is too

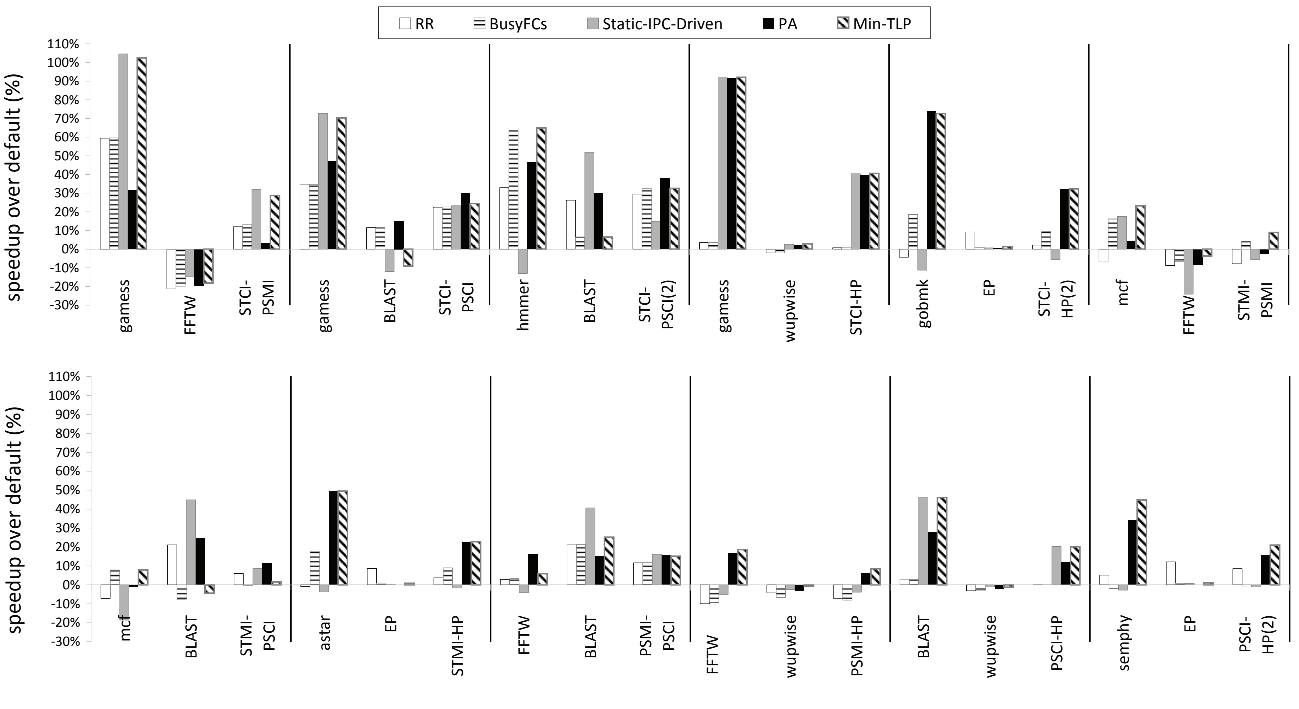


Figure 4: Speedup of asymmetry-aware schedulers on 1FC-12SC for multi-application workload set #1.

high will hide sequential phases to the scheduler. With *spin-then-notify* primitive, choosing the right threshold is much easier: the spin threshold can be safely set at a low value of several hundred thousand or a million iterations, and the blocking threshold can be set at a very high value to avoid performance loss. Because of this flexibility, PA-RTX even outperforms PA-base (adaptive) with the best threshold, by as much as 5% in some experiments.

3.3 Multi-application workloads

Figure 6a shows that even simple asymmetry-aware schedulers trivially accelerate sequential phases and beat the default scheduler when there is only one parallel application running in the system. But as we demonstrate next, they fail to achieve improvements comparable to PA in more realistic multi-application scenarios.

This section shows our results for multi-application workloads. We study the performance of RR, BusyFCs, Static-IPC-Driven, Min-TLP and PA and compare it with Default on 1FC-12SC and 4FC-12SC configurations. Runtime extensions are not used in this case, and the applications are run under the adaptive synchronization mode with the default blocking threshold. Tables 2 and 3 show the two sets of multi-application workloads we used for our evaluation. The workload names in the left column of both tables indicate the class of each application listed in the same order as the corresponding benchmarks, so for example in the STCI-PSMI category `gamess` is the single-threaded compute-intensive (STCI) application and `FFTW` is the partially sequential memory-intensive (PSMI) application. Note that all highly parallel applications have been presented as "HP" without distiction between memory- and CPU-intensive subclasses. Their MI/CI suffix has been removed deliberately to emphasize that schedulers that rely on the number of active threads when making scheduling decisions (Min-TLP and PA) map all threads of HP applications on slow cores regardless of their memory-intensity (either HPCI or HPMI). Nevertheless, the class of each HP application in the workloads can be found in Table 1.

The numbers in parentheses next to each parallel application in Tables 2 and 3 indicate the number of threads chosen for that application. In the first set, the first number corresponds to the 1FC-12SC configuration and the second one to the 4FC-12SC configuration. The workloads from the second set (Table 3) were run on the 4FC-12SC configuration only, so one thread number is included next to each parallel application.

Figures 4 and 5 show the results for the workload set #1 on 1FC-12SC and the workload set #2 on 4FC-12SC configurations, respectively. Results for set #1 on 4FC-12SC are omitted due to space limitations. In each graph, there is a speedup bar for each application in the workload as well as the geometric mean speedup for the workload as a whole, labeled with the name of the workload from Tables 2 and 3. We provide a detailed discussion of performance on the 1FC-12SC scenario (results for the 4FC-12SC configuration are interpreted with similar explanations).

Examining performance results of the simple asymmetry-aware schedulers – BusyFCs and RR– in Figure 4, we see that these algorithms deliver non-negligible speedups over default for workloads with a low number of active threads, i.e., pairs consisting of an ST and a PS application, or two PS applications. When the number of threads is small, the probability that these schedulers map the "most suitable" thread to the fast core is rather high, so their performance is close to more sophisticated schedulers. In cases where HP applications are present, however, PA and MinTLP significantly outperform these simpler schedulers, delivering up

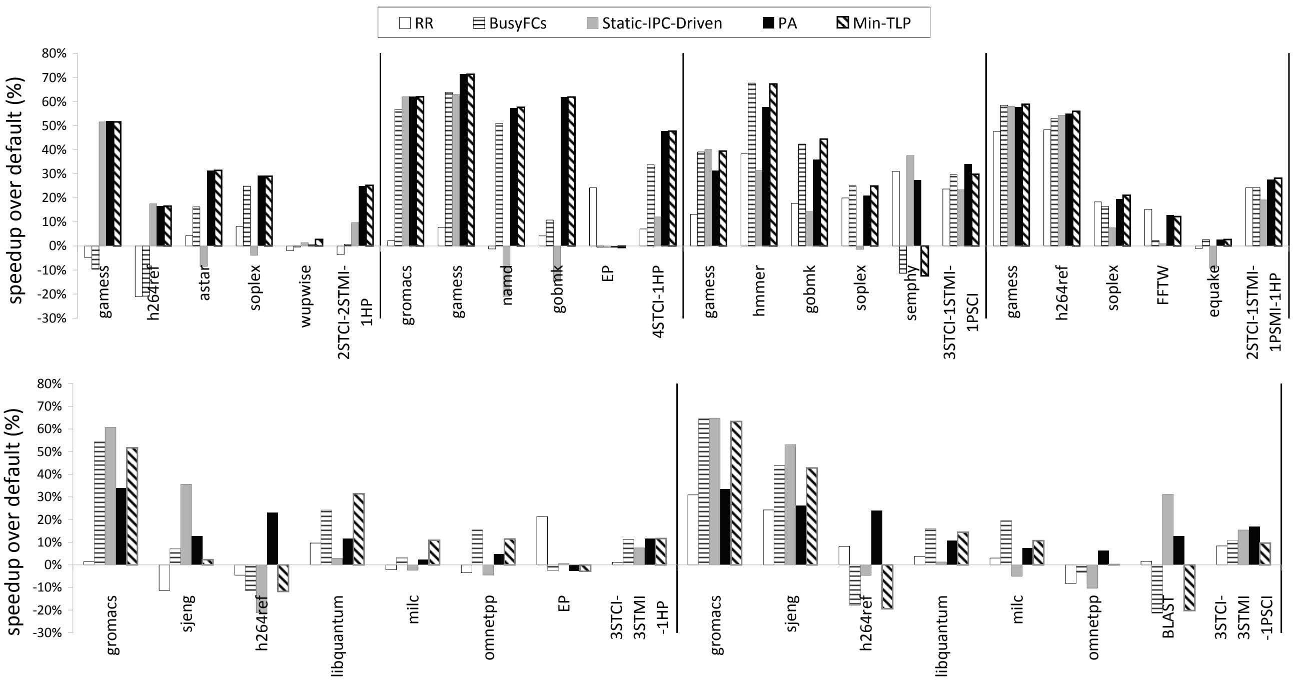


Figure 5: Speedup of asymmetry-aware schedulers on 4FC-12SC for multi-application workload set #2.

to 40% performance improvements (STCI-HP) over them as well as over Default.

MinTLP and PA offer different performance in some cases. Although PA is fairer, because it shares fast cores equally among eligible threads, fairness sometimes comes at a cost. This is especially evident in scenarios with STCI-PSMI and STMI-PSMI workloads, where PA fair-shares the fast core between the ST application and the thread running sequential phase of the PSMI application. Since the PS application is memory-intensive, constantly migrating its *serial* thread between fast and slow cores, which *do not* share a last-level cache in this configuration, hurts performance. At the same time, in workloads consisting of a single-threaded application and a partially sequential compute-intensive application (STCI-PSCI, STCI-PSCI(2) and STMI-PSCI), fair-sharing the fast core enables PA to deliver fairness and even outperform Min-TLP.

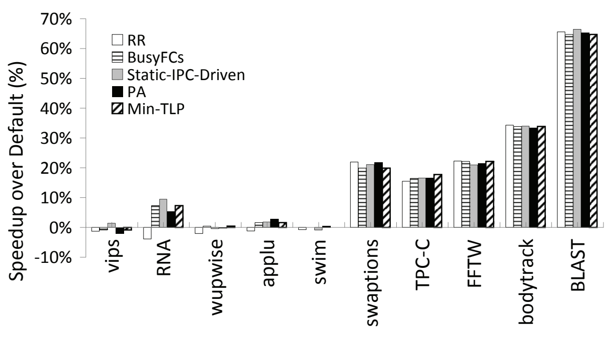
The implication of these results is that migration overhead must be reduced or taken into account if a scheduler is to deliver both performance and fairness. This is addressed by a *migration-friendly* system configuration and a *topology-aware* scheduler design, as will be explained in Section 3.4. Figure 5, for the 4FC-12SC migration-friendly topology, demonstrates this scenario. In this case, many of the costly migrations of threads across memory hierarchy domains (i.e., among cores that do not share a last-level cache) are eliminated, and so PA almost never underperforms MinTLP for the workload as a whole.

We now turn our attention to the Static-IPC-Driven algorithm. Overall, we see that this algorithm performs comparably to PA and MinTLP only in some workloads including an STCI application. In these cases, Static-IPC-Driven assigns the ST thread to the fast core, because it has the highest static IPC ratio due to its compute-bound nature. But because this thread also happens to be the most suitable candidate from the thread-level parallelism stand-point, Static-IPC-Driven makes a decision similar to MinTLP and PA. However, in cases where the least suitable application (HP) is more compute bound, e.g., EP in the STCI-HP(2) workload, Static-IPC-driven makes a wrong decision and performs worse than PA and MinTLP. For the other workloads, PA and MinTLP outperform Static-IPC-Driven. These results once again demonstrate that only parallelism-aware schedulers perform well across the board for a wide range of workloads.

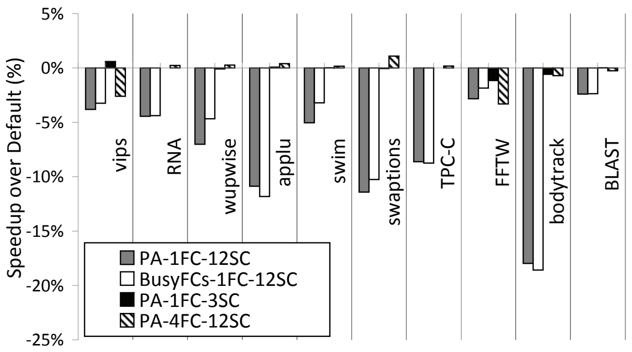
3.4 Evaluating the overhead

Cross-core migrations are an essential mechanism in any asymmetry-aware scheduler. Migrations can be especially costly if the source core is in the different domain of the memory hierarchy than the target core, i.e., the two cores do not share a last level cache. Migration cost is defined as the migration-induced performance loss relative to the Default scheduler. To measure only the performance loss and not the performance improvements resulting from the asymmetry-aware policy we set all the cores in the system to be slow. The asymmetry-aware scheduler, however, still "thinks" that some cores are fast, so it still performs the migrations in accordance with its policies, incurring the cost but not reaping the benefit.

We hypothesized that migration overhead would be mitigated on systems with a *migration-friendly* topology: where at least one fast core would be in the same memory-hierarchy domain with several slow cores, and where the scheduler would avoid cross-domain migrations when possible (*topology-aware*). To evaluate this hypothesis, we study the overhead of PA on several different configurations: 1FC-12SC, 1FC-3SC, and 4FC-12SC. In the first configuration, the fast core



(a)



(b)

Figure 6: (a) Speedups for single-application workloads on 4FC-12SC. (b) Migration overhead.

is in a separate memory hierarchy domain than the slow cores. This is *not* a migration-friendly topology. In the 1FC-3SC configuration only one memory hierarchy domain is used and all cores, fast and slow, are within that domain. This is the most migration-friendly topology. Finally, the 4FC-12SC is a hybrid configuration where each fast core shares a memory hierarchy domain with three slow cores. This topology is complemented by policies that avoid cross-domain migrations whenever possible[3], and so this configuration is also migration-friendly.

Figure 6b shows the overhead of migrations with the PA algorithm on these three configurations. We use a set of parallel applications exposing a wide variety of sequential portion and memory-intensity. The numbers show the relative speedup over Default, so lower numbers mean higher overhead. We also show the overhead for Busy-FCs on 1FC-12SC, to demonstrate that migration overhead is fundamental to asymmetry-aware schedulers in general, not only to PA and MinTLP.

We observe that the highest overheads are incurred on a migration-unfriendly 1FC-12SC topology. On a migration-friendly 1FC-3SC topology, the overheads become negligible. On 4FC-12SC, a more realistic configuration, the migration overheads are also very low.

Our conclusion is that asymmetry-aware policies can be implemented with relatively low overhead on AMP systems, even when it is not possible to ensure that all cores share a single memory hierarchy domain. The key is to design the system such that fast cores share a memory domain with at least some slow cores and, very importantly, to extend the scheduler to avoid cross-domain migrations when possible.

4. RELATED WORK

Previous studies on scheduling for asymmetric multicore systems were united by a common goal of identifying the threads or applications that benefited the most from running on fast cores, but the methods used to identify them were different. One group of researchers, like us, focused on using the amount of parallelism in the application as a heuristic for deciding which threads to run on fast cores. The second group relied on direct measurement or modelling of the application's relative speedup when running on a fast core.

[3]PA, MinTLP as well as our implementations of RR, BusyFCs and Static-IPC-Driven, are topology-aware.

We begin with the discussion of the research in the first group. Hill and Marty [7] and Morad, Weiser and Kolody [12] derived theoretical models for speedup on AMPs for parallel applications with serial phases. These models assumed a scheduler like PA, which maps serial phases to fast cores. These groups demonstrated the benefits of the PA policy theoretically, but the practical issues related to overhead and effectiveness for real applications were not addressed. Our study takes this work further and evaluates the PA policy on a real system. Annavaram et al. [2] designed an application-level scheduler that mapped sequential phases of the applications to fast cores. This scheduler is effective only when the application in which such a scheduler is implemented is the only one running on the system, but not in more realistic scenarios where there are multiple applications. We showed that a scheduler with a global knowledge of the workload, like PA, is necessary to deliver the best performance for multi-application workloads. Furthermore, Annavaram's scheduler required manual changes to the applications, whereas PA delivers its policy to unmodified applications. We also showed that additional speedups are achieved by augmenting threading libraries with PA runtime extensions.

ACS is a combination of hardware and compiler support that was explicitly designed to accelerate lock-protected critical sections on AMPs [17]. Migrations are performed entirely in hardware, so the design of an OS scheduler was not addressed. Furthermore, like Annavaram's work, the system supports only single-application workloads. The authors indicate that operating system assistance would be required to support multi-application workloads. Our work provides support similar to that which would be needed by ACS.

In the second group we find the algorithms that used *relative speedup* to determine which threads/applications to assign to fast cores [5, 9, 16]. Kumar et al. proposed an algorithm that monitored the speedup in the instructions per second (IPS) rate of individual threads on fast cores relative to slow cores and mapped to fast cores those threads that experienced the largest relative speedups [9]. Becchi and Crowley proposed a very similar approach [5]. An algorithm proposed by Shelepov et al. did not require online monitoring of the speedup, but relied on offline-generated architectural signatures for deriving its speedup estimates [16]. None of these schedulers considered thread-level parallelism when mapping threads to cores, and so they are not optimal for workloads that include multi-threaded applications, as we demonstrated in the experimental section. Nevertheless, these algorithms are very effective for workloads consisting of single-threaded applications only, and so we are investigating on algorithms that augment PA with information about relative speedups [15].

In addition to the algorithms that attempted to optimize the use of fast cores by discovering the most "efficient" threads for running on these cores, several simpler asymmetry-aware algorithms were proposed. An algorithm proposed by Balakrishnan et al. ensured that the fast cores do not go idle before slow cores [4] – this is the same approach used by the BusyFC algorithm, which we used in the evaluation. An algorithm proposed by Li et al. also used a BusyFC-like policy, but in addition ensured that the fast cores were given a higher load than slow cores [10]. Neither of these schedulers considered thread-level parallelism when making scheduling decisions.

5. CONCLUSIONS AND FUTURE WORK

In this work we have evaluated the benefits and drawbacks of parallelism-aware scheduling policies for AMP systems. While some algorithms that do not consider TLP perform comparably to the proposed algorithms PA and MinTLP in some scenarios, none of them perform well across the board. PA and MinTLP outperform the other schedulers by as much as 40% in some cases. This indicates the importance of considering TLP in asymmetry-aware scheduling. We have also proposed a small set of runtime extensions to complement the OS scheduler and reduce the possibility of wastefully scheduling busy-waiting threads on fast cores.

Overall, our results have shown that while PA and MinTLP effectively schedule parallel workloads, an asymmetry-aware algorithm that relies on relative speedup brings significant performance improvements for single-threaded applications. Designing a comprehensive scheduler that combines both the TLP and relative speedup to cater to both single-threaded and parallel workloads is an interesting avenue for future work.

Acknowledgements

This research was funded by the Spanish government's research contracts TIN2008-005089 and the Ingenio 2010 Consolider ESP00C-07-20811, by the HIPEAC^2 European Network of Excellence, by the National Science and Engineering Research Council of Canada (NSERC) under the Strategic Project Grant program and by Sun Microsystems. Juan Carlos Saez is supported by a MEC FPU fellowship grant.

6. REFERENCES

[1] T. Anderson. The Performance of Spin Lock Alternatives for Shared-Money Multiprocessors. *IEEE TPDS*, 1(1):6–16, 1990.

[2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI Throttling. In *Proc. of ISCA'05*, pages 298–309, 2005.

[3] D. H. Bailey, E. Barszcz, and J. T. Barton et al. The NAS parallel benchmarks—summary and preliminary results. In *Supercomputing '91*, pages 158–165, 1991.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *SIGARCH CAN*, 33(2):506–517, 2005.

[5] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proc. of Computing Frontiers '06*, pages 29–40, 2006.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of PACT'08*, October 2008.

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

[8] R. Kumar, K. I. Farkas, and N. Jouppi et al. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *Proc. of MICRO 36*, 2003.

[9] R. Kumar, D. M. Tullsen, and P. Ranganathan et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. of ISCA '04*.

[10] T. Li, D. Baumberger, and D. A. Koufaty et al. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proc. of SC '07*, pages 1–11.

[11] T. Li, A. R. Lebeck, and D. J. Sorin. Spin Detection Hardware for Improved Management of Multithreaded Systems. *IEEE TPDS*, 17(6):508–521, 2006.

[12] T. Morad, U. Weiser, and A. Kolody. ACCMP – Asymmetric Cluster Chip Multi-Processing. *TR CCIT*, 2004.

[13] R. Narayanan, B. Ozisikyilmaz, and J. Z. et al. MineBench: A Benchmark Suite for Data Mining Workloads. 2006.

[14] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. of USENIX*, pages 43–43, 1999.

[15] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodourov. A Comprehensive Scheduler for Asymmetric Multicore Systems. In *Proc. of ACM Eurosys '10*, 2010.

[16] D. Shelepov, J. C. Saez, and S. Jeffery et al. HASS: a Scheduler for Heterogeneous Multicore Systems. *ACM Operating System Review*, 43(2), 2009.

[17] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In *Proc. of ASPLOS '09*, pages 253–264, 2009.

[18] R. Teodorescu and J. Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In *Proc. of ISCA '08*, 2008.

[19] R. van der Pas. The OMPlab on Sun Systems. In *Proc. of IWOMP'05*, 2005.