A Comprehensive Scheduler for Asymmetric Multicore Systems

Juan Carlos Saez
Manuel Prieto
Complutense University, Madrid, Spain
{jcsaezal,mpmatias}@pdi.ucm.es

Alexandra Fedorova
Sergey Blagodurov
Simon Fraser University, Vancouver BC, Canada
{fedorova,sba70}@cs.sfu.ca

Abstract

Symmetric-ISA (instruction set architecture) asymmetric-performance multicore processors were shown to deliver higher performance per watt and area for codes with diverse architectural requirements, and so it is likely that future multicore processors will combine a few *fast* cores characterized by complex pipelines, high clock frequency, high area requirements and power consumption, and many *slow* ones, characterized by simple pipelines, low clock frequency, low area requirements and power consumption. Asymmetric multicore processors (AMP) derive their efficiency from core specialization. *Efficiency specialization* ensures that fast cores are used for "CPU-intensive" applications, which efficiently utilize these cores' "expensive" features, while slow cores would be used for "memory-intensive" applications, which utilize fast cores inefficiently. *TLP (thread-level parallelism) specialization* ensures that fast cores are used to accelerate sequential phases of parallel applications, while leaving slow cores for energy-efficient execution of parallel phases. Specialization is effected by an asymmetry-aware thread scheduler, which map threads to cores in consideration of the properties of both. Previous asymmetry-aware schedulers employed one type of specialization (either efficiency or TLP), but not both. As a result, they were effective only for limited workload scenarios. We propose, implement, and evaluate CAMP, a Comprehensive AMP scheduler, which delivers both efficiency and TLP specialization. Furthermore, we propose a new light-weight technique for discovering which threads utilize fast cores most efficiently. Our evaluation in the OpenSolaris operating system demonstrates that CAMP accomplishes an efficient use of an AMP system for a variety of workloads, while existing asymmetry-aware schedulers were effective only in limited scenarios.

EuroSys'10, April 13–16, 2010, Paris, France.

Categories and Subject Descriptors D.4.1 [*Process Management*]: Scheduling

General Terms Asymmetric multicore, Scheduling, Operating Systems

Keywords Asymmetric multicore, Scheduling, Operating Systems

1. Introduction

Asymmetric multicore processors (AMP) [7, 8] were proposed as a more power efficient alternative to symmetric multicore processors (SMP), which use identical cores. An AMP would contain cores that expose the same instruction-set architecture, but differ in features, size, power consumption and performance. A typical AMP would contain cores of two types: "fast" and "slow" [8]. Fast cores are characterized by complex super-scalar out-of-order pipelines, aggressive branch-prediction and pre-fetching hardware, and high clock frequency. Slow cores, on the other hand, have a simple in-order pipeline, less complex hardware, and a lower clock speed. Fast cores occupy a larger area and consume more power than slow cores, so a typical system would contain a small number of fast cores and a large number of slow cores.

An AMP can potentially deliver a higher performance per watt than an SMP [5, 8]. Having cores of different types enables *specializing* each core type for applications that will use it most efficiently. The two most common types of specialization are *efficiency specialization* and *TLP specialization*.

Efficiency specialization Consider a workload consisting of CPU-intensive and memory-intensive applications. The former are characterized by efficient use of CPU pipelines and other "expensive" microarchitectural features. The latter frequently stall the processor as a result of issuing many memory requests, and thus use the CPU inefficiently. Symmetric multicore processors (SMP) will deliver optimal performance/energy trade-off only for one type of applications: an SMP implemented with fast and complex cores would be ideally suited for CPU-intensive applications, while an SMP with slow simple cores will provide better performance/watt for memory-intensive applications [8].

An AMP where both types of cores are present will address applications of both types. It was shown that as a result of core specialization an AMP system can deliver up to 60% more performance per watt than a symmetric one [8].

TLP specialization Consider a workload consisting of parallel and sequential applications, or alternatively of parallel applications with sequential phases. Previous work has shown that it is roughly twice as efficient to run parallel code on a large number of small, slow and low-power cores than on a smaller number of fast and powerful cores, comparable in area and power [5]. The fast cores provide fewer parallel engines per unit of area and power than smaller and simpler slow cores, and so the parallel code will experience worse performance/watt. On the other hand, sequential code will perform poorly on simple slow cores, because it can use only one core at a time. AMP systems offer the best of both worlds. Fast cores can be used for sequential code, while slow cores for parallel code, optimizing performance/watt for code of both types [5].

Specialization must be aided by a thread scheduler that will decide which threads to run on fast cores and which on slow cores. Two kinds of operating system schedulers emerged to address this challenge. The first type targeted efficiency specialization, by assigning the most CPU-intensive threads to fast cores [3, 7, 12]. The second type targeted TLP specialization, by assigning sequential applications and sequential phases of parallel applications to run on fast cores [4]. Both types of schedulers have proved beneficial for their respective target workloads: efficiency specialization delivered benefits for single-threaded workloads, and TLP specialization proved effective for workloads where parallel applications were present. It was not made clear, however, whether it is worth combining these two approaches in a single algorithm and what would be the impact of this comprehensive scheduling solution. In other words, should operating systems for asymmetric processors use an algorithm focusing on efficiency specialization, an algorithm focusing on TLP specialization, or an algorithm that performs both types of specialization? The goal of our study is to address this question.

To that end, we propose CAMP, a new Comprehensive AMP scheduling algorithm that delivers both types of specialization. To the best of our knowledge, this is the first asymmetry-aware algorithm addressing this goal. The challenge in implementing this algorithm is equipping it with an effective mechanism for deciding which threads are more "profitable" candidates for running on fast cores. To that end, we introduce the new metric *Utility Factor* (UF), which accounts for both efficiency and TLP of the application and produces a single value that approximates how much the application as a whole will improve its performance if its threads are allowed to occupy all the fast cores available on that system. The utility factor is designed to help the scheduler pick the best threads to run on fast cores in non-trivial cases, such as the following. Consider a workload of a CPU-intensive application with two runnable threads and a less CPU-intensive with a single thread. In this case, it is not immediately clear which thread is the best candidate for running on the fast core (assuming there is only one fast core on the system). On the one hand, dedicating the fast core to a thread of a two-threaded application may bring smaller performance improvements to the application as a whole than dedicating the fast core to the single-threaded application, because a smaller part of the application will be running on fast cores in the former case. On the other hand, the two-threaded application is more CPU-intensive, so running it on the fast core may be more profitable than dedicating the fast core to another, less CPU-intensive, application. By comparing utility factors across threads the scheduler should be able to identify the most profitable candidates for running on fast cores.

In this work, we use the utility factor to optimize *system-wide* performance. However, in cases where some applications have a higher priority than others or in scenarios where the system needs to deliver QoS guarantees to certain applications, the utility factor could be used as a *complementary* metric to find a balance between providing better service for prioritized applications and maintaining overall performance. For example, if the system determines that a high-priority application has a low utility factor, meaning that little or no speedup would be gained for that application if it or some of its threads were to run on fast cores, then there is no point in "wasting" a fast core on this application, despite its high priority. As a result, the utility factor would be used to enable QoS guarantees with a minimal effect on performance.

Another contribution of our work is the new method for determining the efficiency of a specific thread in using a fast core. Efficiency is measured as the *speedup factor* that the thread experiences from running on a fast core relative to a slow core. Previous approaches for determining the speedup factor had limitations [3, 7, 12]. They either required to run each thread on each core type [3, 7], which caused load imbalance and hurt performance [12], or relied on static offline data [12]. Our new method uses online measurements of last-level cache (LLC) miss rate to inexpensively estimate the speedup factor. The advantage of this method is that it is very efficient and easy to use in a scheduler, and it does not require to run each thread on cores of both types. While our new method does not approximate the speedup factor with a high accuracy, since the model was made deliberately simple for portability across hardware platforms and for efficiency when used in a scheduler, it rather successfully categorizes applications into three classes – low, medium and high – according to their efficiency. As a result, the scheduler using dynamically estimated speedup factors performs within 1% of the oracular scheduler that uses *a priori* known, and thus highly accurate, overall speedup factors.

We implemented CAMP in the OpenSolaris operating system and evaluated it on real multicore hardware where asymmetry is emulated by setting the cores to run at different frequencies via DVFS (dynamic voltage and frequency scaling). We use CPU-bound scientific applications, and construct workloads containing various applications: CPU-intensive and memory-intensive, sequential and parallel, as well as parallel with sequential phases.

We compare CAMP with several other asymmetry-aware schedulers including Parallelism-Aware (PA), which performs only TLP specialization, our new implementation of a Speedup-Factor Driven (SFD) scheduler, which takes into account only efficiency, and a baseline round-robin (RR) scheduler that simply shares fast cores equally among all threads. We find that for workloads consisting exclusively of single-threaded applications, the algorithm focusing only on efficiency specialization is sufficient, but this algorithm is ineffective for workloads containing parallel applications. Conversely, an algorithm focusing only on TLP specialization is effective for workloads containing parallel applications, but not for those where only single-threaded applications are present. CAMP, on the other hand, effectively addresses both types of workloads. We also find that there is some extra benefit in using efficiency specialization in addition to TLP specialization for realistic workloads containing parallel applications. The greatest benefit of CAMP, therefore, is that it optimizes scheduling on AMPs for a variety of workloads, smoothly adjusting its strategy depending on the type of applications running on the system.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes how we compute the utility factor. Section 4 presents the design of the CAMP algorithm and briefly describes other algorithms that we use for comparison. Section 5 describes the experimental results, and Section 6 summarizes our findings.

2. Background and Related Work

Several asymmetry-aware schedulers were proposed in previous work. They delivered either efficiency or TLP specialization, but not both.

Kumar et al. [7] and Becchi et al. [3] independently proposed similar schedulers that employed efficiency specialization. For deciding which applications would use fast cores most efficiently, these schedulers relied on a speedup factor, which is the improvement of an application's performance on a fast core relative to a slow core. The speedup factor was directly measured online by running each thread on cores of both types and computing the desired performance improvement[1]. Algorithms proposed by Becchi and Kumar were evaluated in a simulated environment. When real implementations of these algorithms were done as part of our earlier work, we found that the proposed methods for computing the speedup factor were inaccurate since applications may exhibit a non uniform behavior during and between program phases. Only if applications have a stable behavior, observations over time provide satisfactory speedup factor estimations. Furthermore, observation on both core types requires additional thread migrations that can cause significant performance degradation and load imbalance [12]. To address these problems, we proposed an alternative method for obtaining the speedup factor, which involved estimating it using static information, an *architectural signature* embedded in the application binary. An architectural signature contains information enabling the scheduler to estimate the application's last-level cache miss rate and with reasonable accuracy predict its speedup factor on cores of different types. Although this method, used in our Heterogeneity-Aware Signature Supported (HASS) scheduler, overcame the difficulties associated with the direct measurement of the speedup factor [12], it had limitations of its own. In particular, it relied on static information about the application obtained offline, which did not always allow to capture dynamic properties of the application and required co-operation from the developer to perform the steps needed for the generation of the architectural signature. To overcome this shortcoming, we designed a new method for estimating the speedup factor: one that could be easily used online and would introduce negligible performance overhead. This method is described in Section 4.1.2.

TLP specialization was employed in our earlier algorithm called Parallelism-Aware (PA) [4] – this is the only such operating system algorithm of which we are aware. PA used the number of runnable threads as the approximation for the amount of parallelism in the application [4]. We found that this was indeed a good heuristic for the approximation, because applications typically let the unused threads block, perhaps after a short period of spinning. Blocking is coordinated with the operating system, so a scheduler can detect the change in the TLP and adjust its scheduling strategy appropriately. An exception is idle threads that busy-wait on a CPU, remaining runnable even though they do not make useful progress. However, most synchronization libraries implement adaptive synchronization algorithms where threads only busy-wait for short periods of time and then block. In addition, a *spin-then-notify* synchronization mode was also proposed in our study to make spinning visible to the operating system. Since using runnable thread count proved to be a good heuristic for approximating the amount of parallelism in the application, we use it along with other metrics for computing the utility factor in the CAMP scheduler.

Prior to our work on the PA algorithm, Annavaram et al. proposed a application-level AMP algorithm that caters to the application's TLP [1], but this algorithm required modifying an application whereas our approach is able to improve performance in a transparent way. Furthermore, an application-level scheduler only addresses scenarios when

[1] Only single-threaded applications were evaluated in that work.

there is only one application running in the system, while an OS scheduler addresses multi-application workloads as well.

A scheduler proposed by Mogul et al. employed another type of core specialization (not previously discussed in this paper) where one slow core was reserved for executing system calls [11]. In a similar vein, Kumar et al. proposed specializing a slow core for running the controlling domain of a virtual machine monitor Xen [9]. Unfortunately performance improvements from Mogul's scheduler were not very large due to overhead associated with migrating a system call to a slow core. Nevertheless, our implementation is also aware of this kind of core specialization since all Solaris' kernel threads are scheduled on slow cores.

Other asymmetry-aware schedulers of which we are aware did not target core specialization, but pursued other goals, such as ensuring that a fast core does not go idle before slow cores [2], or keeping the load on fast cores higher than the load on slow cores [10].

While existing AMP schedulers addressed parts of the problem they did not provide a comprehensive solution: one that would address a wide range of workloads as opposed to targeting a selected workload type. Our goal in CAMP scheduler is to close this gap.

Finally, we should highlight that in this work we emulated an asymmetric system by scaling the frequency of individual cores. Although this is not the most accurate way to approximate future AMP platforms, in which the cores of different types are also likely to have different pipeline architectures (e.g., simple in-order vs. complex out-of-order), this methodology permitted us to run our experiments on a real system, as opposed to a simulator. As a result, we were able to do a much more extensive analysis than what would have been possible on a simulator. Nevertheless, work by other researchers (carried out concurrently with ours) suggests that conclusions made in our work would apply to asymmetric systems with more profound differences between core micro-architectures. We are referring to the work by Koufaty, Reddy and Hahn from Intel [6], where the authors used proprietary tools to emulate an asymmetric system where the cores differed in the number of micro-ops that could be retired per cycle. The authors assumed cores of two types: a *big* core capable of retiring up to four micro-ops per cycle, and a *small* core capable of retiring at most one micro-op per cycle. For single threaded applications from the SPEC CPU 2006 suite, they found that the relative speedup on the big core relative to a small core highly correlates with the amount of *external stalls* generated by the application, which are in turn approximated by memory reads and requests for cache line ownership. In our work, this relative speedup is approximated using last-level cache miss rates, which include the metrics used in the Intel work and would have a high correlation with them. In conclusion, this suggests that the findings of our work could have direct application to systems with more significant differences between the cores than in our experimental system.

3. Utility Factor

Given a system with N_{FC} fast cores, the *Utility Factor* (*UF*) is a metric approximating the application speedup if N_{FC} of its threads are placed on fast cores and any remaining threads are placed on slow cores, relative to placing all its threads on slow cores. Speedup is measured using the following formula: $Speedup = T_{base}/T_{alt}$, where T_{base} is the completion time for the application in the "base" configuration, where only slow cores are used, in our case, and T_{alt} is the completion time in the "alternative" configuration, where both fast and slow cores are used.

The formula for the *UF* is shown in Equation 1:

$$UF = \frac{SF_{app}}{MAX(1, N_{THREADS} - (N_{FC} - 1))^2} \quad (1)$$

$N_{THREADS}$ is the number of threads in the application, which is visible to the operating system, since most modern runtime environments map user threads one-to-one onto kernel threads. SF_{app} is the average speedup factor of the application's threads when running on a fast core relative to a slow core; we describe how we obtain it at runtime in Section 4.1.2.

In constructing the model for the utility factor we make two simplifying assumptions:

- Only the threads of the target application, i.e., the application for which the *UF* is estimated, are allowed to use fast cores. This would not be the case under a fair scheduling policy, which would attempt to share fast cores among all "eligible" threads from different applications. Taking into account all possible ways in which fast cores can be shared, however, would introduce too much complexity for an efficient online algorithm.
- The number of threads in the application does not exceed the number of slow cores. Given that the number of slow cores is likely to be large relative to fast cores, this assumption is, first of all, reasonable, because CPU-bound applications are not likely to be run with more threads than cores [13], and, second, will not introduce a significant error into our model, at least for the applications that we considered (parallel scientific applications from SPEC OMP2001 suite, plus a few others). As the number of threads begins to exceed the number of fast cores, the *UF* rapidly approaches zero. So even if the application has more threads than assumed by our formula, the *UF* estimation should remain accurate.

The scheduler will use this model to estimate the utility factor for each application. Threads of the application with the highest utility factor will then be assigned to run on fast cores: the higher the utility factor the more the application

benefits from using fast cores. Note that if an application dynamically changes the number of active threads, as it enters a sequential phase, for example, the scheduler would recompute the utility factor to reflect this change.

Let us describe the intuition behind the formula for the utility factor. The easiest way to understand it is to first consider the case where the application has only a single thread. In this case, *UF*=*SF*; in other words the utility factor is equal to the speedup that this application will experience from running on a fast core relative to a slow core. Next, let us address the case when the application is multithreaded. If all threads were running on fast cores, then the entire application would achieve the speedup of *SF*. In that case, the denominator is equal to one and *UF*=*SF*. However, if the number of threads is *greater* than the number of fast cores, then only *some* of the threads will run on fast cores and the overall utility factor will be less than *SF*. To account for that, we must divide SF by one greater than the number of threads that would *not* be running on fast cores: $N_{THREADS} - (N_{FC} - 1)$. Finally, we introduce a quadratic factor in the denominator, because we determined experimentally that if some of the threads experience the speedup because of running on fast cores and others do not, the overall application speedup is smaller than the portion of speedup achieved by threads running on fast cores. That is because threads running on fast cores must synchronize with the threads running on slow cores, so they do not fully contribute to the application-wide speedup. Introducing the quadratic factor in the formula enables to account for that effect rather accurately as we will show with experimental results.

Next we demonstrate that the utility factor model closely approximates the speedup on asymmetric systems for highly parallel applications. Figure 1 shows the estimated and actual *UF* for several parallel applications with different synchronization patterns and memory-intensity using an asymmetric-aware scheduler that keeps all fast-cores busy[2]. We performed validation for other highly parallel applications from the OpenMP2001 and Minebench suites as well, but the data is omitted due to space limitations, especially since they behave similarly. The *UF* was measured and estimated for a machine with two fast cores and eight slow cores (2FC-8SC). A fast core was twice as fast as the slow core (we provide more details on our experimental setup in Section 5). The *SF* for the threads was estimated offline by running the application with a single thread first on slow cores, then on fast cores and computing the speedup. The figure shows that the estimated utility factor closely tracks the quantity it attempts to approximate.

[2] This scheduler (described in [10]) resembles the actual behavior of our CAMP scheduler when a highly parallel application runs solo in the system.

4. Design and Implementation

In this section we describe the design and implementation of the CAMP scheduler as well as the other schedulers used for comparison.

4.1 The CAMP Scheduler

4.1.1 The algorithm

CAMP decides which threads to place on cores of different types based on their individual utility factors. According to their utility factors, threads are categorized into three classes: LOW, MEDIUM, and HIGH. Using classes allows to mitigate any inaccuracies in estimation of *SF* used in the *UF* formula as well as provide comparable treatment for threads whose utility factors aver very close. Threads falling in the HIGH utility class will run on fast cores. If there are more such threads than fast cores, the cores will be shared among these threads equally, using a round-robin mechanism.

If after all high-utility threads were placed on fast cores there are idle fast cores remaining, they will be used for running medium-utility threads or, if no such threads are available, low-utility threads (we do not optimize for power consumption in this work, so our scheduler tries to keep fast cores as busy as possible). In contrast with threads in the HIGH utility class, fast cores will not be shared equally for threads in the MEDIUM and LOW utility classes. Sharing the cores equally implies cross-core migrations as threads are moved between fast and slow cores. These migrations hurt performance, especially for memory-intensive threads, because threads may lose their last-level cache state as a result of migrations. The effect of migrations in asymmetry-aware schedulers on performance was extensively explored in our previous work [4], and so we do not provide the related performance analysis in this paper.

Threads of parallel applications executing a sequential phase will be designated to a special class SEQUENTIAL_BOOSTED. These threads will get the highest priority for running on fast cores: this provides more opportunities to accelerate sequential phases. Only high-utility threads, however, will be assigned to the SEQUENTIAL_BOOSTED class. Medium- and low-utility threads will belong to their regular class despite running sequential phases. Since these threads do not use fast cores efficiently, it is not worthwhile to give them an elevated status. Threads placed in the SEQUENTIAL_BOOSTED class will remain there for the duration of `amp_boost_ticks`, a configurable parameter. After that, they will be downgraded to their regular class, as determined by the utility factor, to prevent them from monopolizing the fast core.

The class-based scheme followed by CAMP relies on two utility thresholds, lower and upper, which determine the boundaries between the LOW, MEDIUM and HIGH utility classes. The lower threshold is used to separate the LOW and

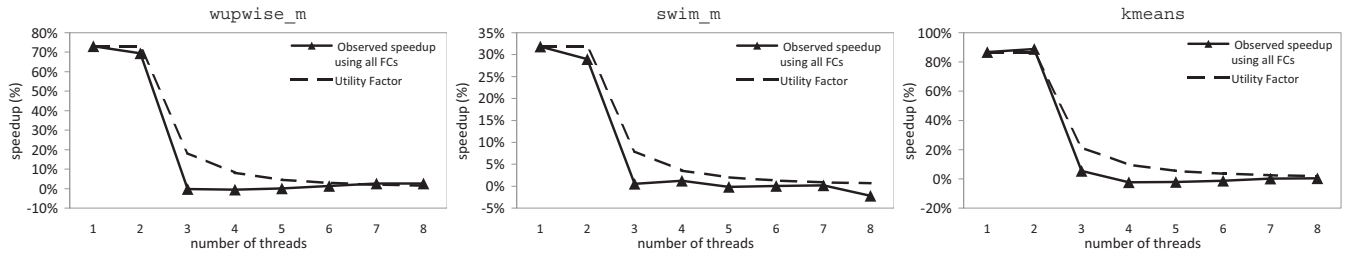

Figure 1. Comparison between the observed speedup using all fast cores in a 2FC-8SCs configuration over using slow cores only to the speedup approximated using the *UF* model.

MEDIUM classes, the upper threshold is used to separate the MEDIUM and HIGH classes.

CAMP has a built-in mechanism to dynamically, and in a transparent way, select which utility thresholds to use based on the system workload. There are two pairs of utility thresholds, one for when only single-threaded applications run on the system, and the other for when at least one multi-threaded application is running on the system. When multi-threaded applications are present, we see a higher range of utility factors than when only single-threaded applications are present, and so different thresholds than in single-threaded mode are used to reflect this higher range. These thresholds are also machine dependent: on systems with a large difference between the speed of fast and slow cores the utility factors will be larger than on systems where this difference is small.

For example, utility factors for single-threaded applications used in our study were 23 and above on our experimental system (because the speedup factor relative to a slow core is at least 23%). At the same time, multi-threaded applications will often have a utility factor as low as 0 (Figure 1). So a single-threaded application whose utility factor is low relative to other single-threaded applications will nevertheless have a high *UF* relative to most multi-threaded applications. To properly reflect the relationship between threads' *UF*s when placing them into classes we use different sets of thresholds for single-threaded and multi-threaded scenarios.

4.1.2 Computing the speedup factor

A speedup factor for a thread is formally defined as $\frac{IPS_{fast}}{IPS_{slow}}$, where IPS_{fast} and IPS_{slow} are the thread's instructions per second (IPS) ratios achieved on fast and slow cores respectively. As explained earlier, the traditional method for computing the *SF* involved running each thread on each core type, but that disrupted load balance and hurt performance.

Our new method for computing the speedup factor *SF* relies on threads' LLC (last-level cache) miss rates measured online using hardware performance counters. The miss rate can be measured on any type of core; there is not need to run the thread on both core types. To estimate the *SF* from the LLC miss rate we used the following approach developed in our previous work [12]. We compute the hypothetical completion time for some constant number of instructions on both core types. We compose the completion time of two components: execution time and stall time. To compute the execution time we assume a cost of 1.5 cycles per instruction and factor in the clock speed. To compute the stall time, we estimate the number of cycles used to service the LLC misses occurring during that instruction window: for that we require the per-instruction LLC miss rate, which the scheduler measures, and the memory latency, which can be discovered by the operating system.

This method for estimating the stall time abstracts many details of the microarchitecture: the fact that not all cache misses stall the processor because of out-of-order execution, the fact that some cache misses are actually pre-fetch requests that also do not stall the processor, the fact that some cache misses can be serviced in parallel, and the fact that the memory latency may be different depending on memory bus and controller contention as well as non-uniform memory access (NUMA) latencies on some architectures. Accounting for all these factors is difficult, because their complex inter-relationship is not well understood. Using instead a simple model that relies solely on the LLC and assumes a stable latency did not prevent our scheduler from performing successfully (see validation results in Section 5.1). Nevertheless, there is no limitation in CAMP that prevents the use of more accurate *SF* models. For instance, in [6], the authors use a similar approach for estimating *external stalls* (i.e. any stall due to resources external to the core), but their *SF* model also uses additional performance counters to account for *internal stalls* caused by branch mispredictions and the contention of other internal resources. We could extend CAMP with this additional metric but we found it does not provide higher accuracy on our emulated AMP system since internal stalls are the same on both core types (they have the same micro-architecture).

In CAMP, LLC miss rates are measured for each thread continuously, and the values are sampled every 20 timer ticks (roughly 200ms on our experimental system). We keep a moving average of the values observed at different periods and we discard the first values collected immediately after

the thread starts or after it is migrated to another core in order to correct for cold-start effects causing the miss rate to spike intermittently after migration. We also use a carefully crafted mechanism to filter out transitions between different program phases. Updating *SF* estimations during abrupt phase changes may trigger premature changes in the *UF* and, as a result, unnecessary migrations, which may cause substantial performance overhead. Instead, *SF* estimations are updated exclusively once a thread enters a phase of stable behavior. To detect those stable phases, we use a light-weight mechanism based on a *phase transition threshold* parameter (12% in our experimental platform). When the moving average is recorded, it is compared with the previous average measured over the previous interval. If the two differ by more than the transition threshold, a phase transition is indicated. Two or more sampling intervals containing no indicated phase transition signal a stable phase.

On processors with shared caches the thread's miss rate may vary due to the sharing of the cache with other threads, in addition to reasons related to internal program structure. For example, the miss rate may decrease because of cooperative data sharing or increase because of cache contention. However, we observed that the *quality* of the miss rate does not change significantly regardless if the thread shares a cache or runs solo: i.e., if the thread's miss rate is low relative to other threads when it runs solo, its value relative to other threads will stay low when it shares the cache even though it may increase by tens or hundreds of percent relative to its solo value. Similarly, if the thread's miss rate is high it will stay high relative to other threads, regardless if there is sharing.

We define three categories for the speedup factors and each category is labeled by a "representative" *SF* of that category. The representative *SF* is machine-specific and was set empirically. The thresholds delimiting the categories were also chosen experimentally. After estimating a thread's *SF* we determine what category it fits in and assign it the *SF* equal to the label value corresponding to that category. In Section 5 we compare observed and estimated ratio and show that our model have enough accuracy to effectively guide scheduling decisions.

When computing the utility factor for a thread, we do not average the *SF*s of all threads in this application, but we use the *SF* of the thread in question. Averaging the *SF* values would require cross-thread communication, which could damage the scalability of the scheduler. Using the current thread's *SF* is a good approximation of averaging for the following reason. First of all, in most applications we examined (see more about our selected benchmarks in Section 5) all threads do the same type of work, so their *SF* values would be the same. In applications where threads do different work, the most frequently occurring *SF* values will dominate and ultimately determine where most application is scheduled.

Finally, we describe an optimization related to the computation of the utility factor. The overhead of measuring the LLC miss rates is negligible at the sampling rate we use. However, we found during early development stages that computing the *UF* and updating the associated data structures at every sampling period may introduce some overheads. Fortunately, these can be substantially removed by applying certain optimizations. For instance, if we determine that a thread of a highly threaded application could never achieve a MEDIUM or HIGH utility factor even if it had the highest *SF* possible (i.e. the speed ratio between the fast and the slow cores), we do not recalculate the *SF* for the threads in this application unless the number of threads decreases, effectively removing the associated overheads.

4.2 The other schedulers

There are three other schedulers with which we compare the CAMP algorithm: Parallelism-Aware (PA), which delivers TLP specialization only, SF-Driven (SFD), which delivers efficiency specialization only, and round-robin (RR), which equally shares fast and slow cores among all threads. We implemented all these algorithms in OpenSolaris. We do not compare with the default scheduler present in our experimental operating system, because the performance with this scheduler exhibited a high variance making the comparison difficult. Nevertheless, RR's performance is comparable to or better than the default scheduler, so this is a good baseline.

The PA scheduler has the same code base as the CAMP scheduler, but since it accounts only for TLP, it uses the default *SF* value in the *UF* formula (Equation 1). The default *SF* is the upper bound on the achievable *SF* on the given system: the theoretical maximum for the IPS ratios between fast and slow cores. PA, like CAMP, boosts the fast-core priority of threads executing sequential phases of parallel applications by assigning them into SEQUENTIAL_BOOSTED class. However, since PA does not compute applications-specific speedup factor, it cannot distinguish between HIGH, MEDIUM and LOW utility threads. So unlike CAMP, which will place only high-utility threads in the SEQUENTIAL_BOOSTED class, PA will place all threads executing sequential phases in that class.

SFD, similarly to PA uses the *UF* formula where the number of threads is always equal to one, since it does not account for TLP of the application. The SFD estimates the *SF* using our new method based on LLC miss rates.

The RR algorithm shares fast and slow cores among threads using the same mechanism that CAMP uses to share fast cores among applications of the HIGH utility class.

4.3 Topology-aware design

An important challenge in implementing any asymmetry-aware scheduler is to avoid the overhead associated with migrating threads across cores. Any asymmetry-aware scheduler relies on cross-core migrations to deliver the benefits

of its policy. For example, CAMP must migrate a high utility thread from a slow core to a fast core if it detects that the thread is executing a sequential phase. Unfortunately, migrations can be quite expensive, especially if the source and target cores are in different *memory domains* of the memory hierarchy[3]. On NUMA architectures remote memory accesses further aggravate this issue and migration cost can be even higher.

However, any attempt to reduce the number of migrations may backfire by decreasing the overall benefits of asymmetric policies. A more feasible solution, in our opinion, is to consider ways of making migrations less expensive. In particular, if AMP systems are designed such that there is a fast core in each memory hierarchy domain (i.e., per each group of slow cores sharing a cache), migration overhead might be mitigated. Indeed, in a previous paper we have shown [4] that the overhead of migrations becomes negligible with such *migration-friendly* designs as long as the schedulers minimize cross-domain migrations. Based on these insights, our implementations of all the investigated schedulers have been carefully crafted to avoid cross-domain migrations when possible (i.e. all the schedulers are *topology-aware*).

5. Experiments

The evaluation of the CAMP algorithm was performed on an AMD Opteron system with four quad-core (Barcelona) CPUs. The total number of cores was 16. 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 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. Each core is capable of running at a range of frequencies from 1.15 GHz to 2.3 GHz. Since each core is within its own voltage/frequency domain, we are able to vary frequency for each core independently. To create an AMP configuration we configure some cores to run at 2.3 GHz (*fast* cores) and others to run at 1.15 GHz (*slow* cores). We also varied the number of cores in the experimental configurations by disabling some of the cores. For validating accuracy of our method for estimating the speedup factor we also used an Intel Xeon system with two quad-core CPUs, running in the frequency ranges of 2 and 3 GHz.

In our experiments we used four AMP configurations: (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, and (3) 2FC-2SC – two fast cores, two slow cores, each on its own chip .

We experimented with applications from the SPEC OMP 2001, the SPEC CPU 2006, and the Minebench suites, as well as BLAST – a bioinformatics benchmark – and FFT-W – a scientific benchmark performing the fast Fourier transform. In all workloads (multi-application), we ensure that all applications are started simultaneously and when an application terminates it is restarted repeatedly until the longest application in the set completes three times. The observed standard deviation was negligible in most cases (so it is not reported) and where it was large we restarted the experiments for as many times as needed to guarantee that the deviation reached a low threshold. Average completion time for all the executions of a benchmark under a particular asymmetry-aware scheduler is compared to that under RR, and percent 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 CPU-bound workloads that we considered [13].

Our evaluation Section is divided into three parts. In Section 5.1 we evaluate the accuracy of our method for estimating the speedup factor. In Section 5.2 we describe the workloads that we tested and briefly discuss results for single-threaded applications. In Section 5.3 we present aggregate results for all workloads with all schedulers, and analyze the multi-threaded workloads in more detail.

5.1 Accuracy of SF estimation

In this subsection we compare the estimated *SF* to the actual *SF* for all applications in SPEC CPU2006. Actual *SF* is measured by running the application on the slow core, then on the fast core, and computing the speedup. Estimated *SF* is obtained from the average LLC measured throughout the entire run of the application. Figure 2 shows the measured and the estimated ratios on our AMD and Intel systems respectively. Measured speedup ratios obtained in the environment where threads are periodically migrated between different cores are also measured – these data better reflect the realistic conditions under which the *SF* must be obtained

As Figure 2 shows, the estimates are accurate for CPU-intensive applications on both platforms (on the right side of the chart), but less accurate for medium applications (on the center of the chart). As we explained earlier, inaccuracies occur as a result of the simplifying assumptions made in our model.

We observed that model inaccuracies are mitigated when it is used in the scheduler, because the scheduler categorizes applications into coarse Speedup Factor classes rather than relying solely on *SF* estimates. On both platform, the thresholds that define these classes are set empirically to $\frac{1}{2}$ and $\frac{4}{5}$ of the maximum SF attainable, as shown in Figure 2.

5.2 Workloads

We experimented with two sets of workloads: those consisting of single-threaded applications, typically targeted by algorithms like SFD, and those including multi-threaded applications, typically targeted by algorithms like PA.

[3] A memory hierarchy domain in this context is defined as a group of cores sharing a last-level cache.

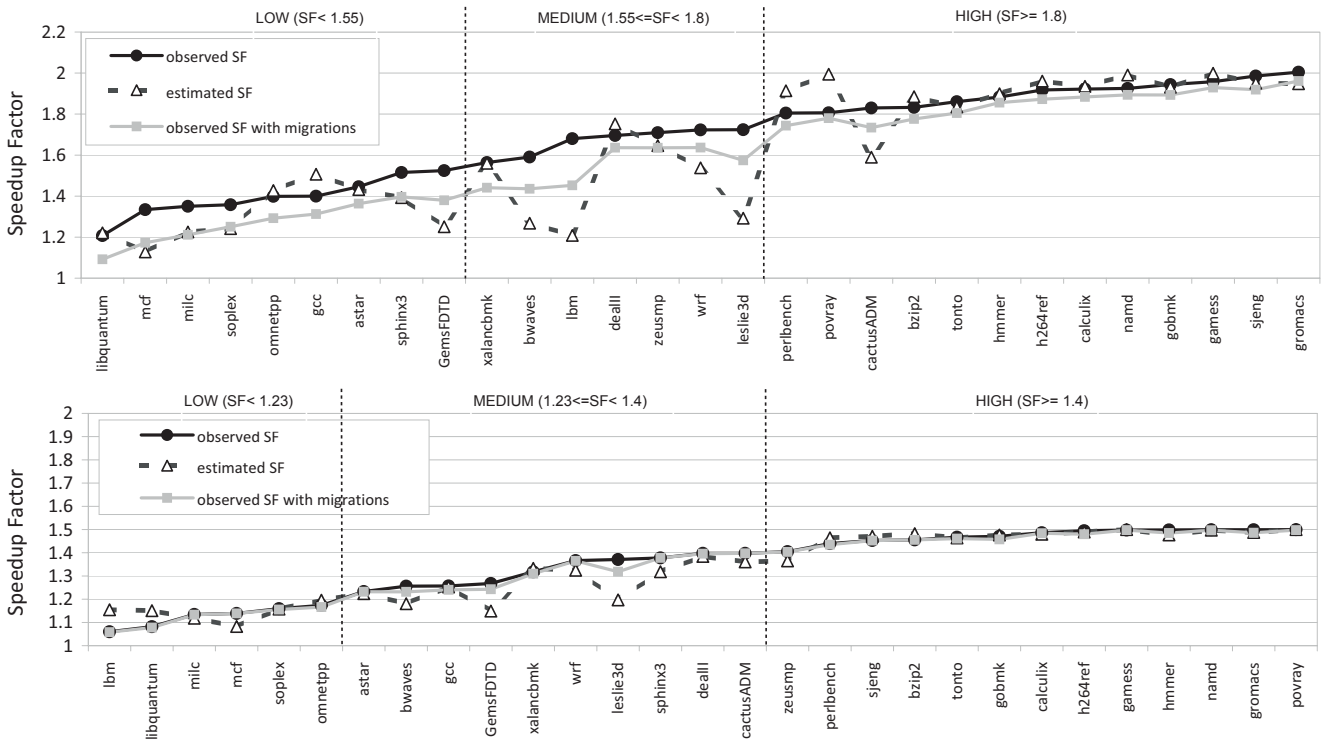

Figure 2. Observed and predicted speedup factors for all benchmarks of SPEC CPU2006 benchmarks on an AMD Opteron (top) and an Intel Xeon (bottom) platforms.

5.2.1 Single-threaded applications

To evaluate our scheduling algorithms for different types of applications and workloads, we selected eleven applications from the SPEC CPU 2006 suite and constructed ten workloads containing representative pairs. In selecting applications, we tried to cover a wide variety of behaviors. Some benchmarks are either memory-intensive (such as `mcf` and `milc`) or CPU-intensive (such as `gromacs` and `sjeng`), whereas others exhibit different phases across their execution (`astar` is a memory-intensive application that also exhibits some cpu-intensive phases).

The ten workloads shown in Table 1 cover a rich set of scenarios. 4CI and 4MI are homogeneous workloads that combine applications of the same class (either CPU-intensive or memory-intensive applications) and xCI-yMI are heterogeneous workloads that mix memory-intensive and CPU-intensive applications. The categories in the left column are listed in the same order as the corresponding benchmarks, so for example in the 1CI-3MI category `gromacs` is the CPU-intensive (CI) application and `milc`, `soplex` and `mcf` are the memory-intensive (MI) applications. The last three workloads labeled as Phased include applications that do not fall into a clean class since they exhibit different phases.

Results for these workloads running under PA, CAMP and SF are shown in Figure 3. To complement our assessment on the effectiveness of SF predictions, we also provide a comparison with a "Best Static" assignment, which ensures applications with the highest overall ratios to run on fast cores. As expected, PA behaves like RR since it is unaware of the efficiency of individual threads and, as a result, assigns all applications to the HIGH utility class and fair-shares fast cores among them. CAMP and SFD perform similarly, since *UF=SF* for single-threaded applications. Overall, we observed that these algorithms effectively distinguish between CPU-intensive and memory-intensive code and perform thread-to-core mappings closer to the "Best Static", in the absence of phase changes (on the Intel platform, SFD and CAMP behave better due to the higher accuracy of the SF

Categories	Benchmarks
4CI	`gamess`, `perlbench`, `povray`, `gromacs`
3CI-1MI	`sjeng`, `gamess`, `gromacs`, `soplex`,
2CI-2MI_A	`perlbench`, `povray`, `soplex`, `mcf`
2CI-2MI_B	`gromacs`, `sjeng`, `milc`, `soplex`
1CI-3MI_A	`gamess`, `milc`, `soplex`, `mcf`
1CI-3MI_B	`gromacs`, `milc`, `soplex`, `GemsFDTD`
4MI	`GemsFDTD`, `milc`, `soplex`, `mcf`
Phased1	`astar`, `astar`, `milc`, `leslie3d`
Phased2	`sjeng`, `astar`, `milc`, `leslie3d`
Phased3	`astar`, `astar`, `GemsFDTD`, `GEMSFDTD`

Table 1. Multi-application workloads consisted of single-threaded applications

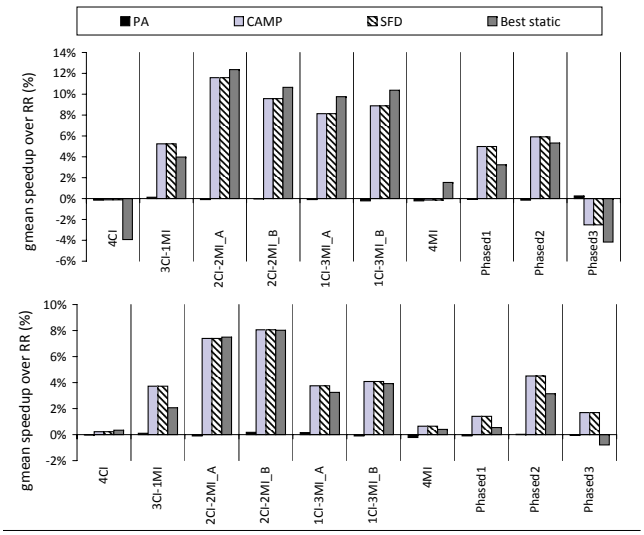

Figure 3. Speedup of PA, SFD, CAMP and Best Static schedulers when running single-threaded workloads on the 2FC-2SC AMD (top) and Intel (bottom) platforms.

estimations). For applications that exhibit different phases across their execution, "Best Static" does not guaranty optimal mappings.

5.2.2 Single-threaded and multi-threaded applications

We categorized applications into three groups with respect to their parallelism: highly parallel applications (HP), partially sequential (PS) applications (parallel applications with a sequential phase of over 25% of execution time), and single-threaded applications (ST). In order to cater to application memory-intensity we divided, in turn, the three aforementioned groups into memory-intensive (MI) and CPU-intensive classes (CI), resulting in six application classes: HPCI and HPMI classes for highly parallel applications, CPU-intensive and memory-intensive, respectively; PSCI and PSMI classes for partially sequential applications, and STCI and STMI classes for single-threaded applications.

We constructed nine workloads consisting of representative pairs of benchmarks across the previous categories mentioned above as shown in Table 2. The categories in the left column are listed in the same order as the corresponding benchmarks, so for example in the STCI-PSMI category `gamess` is the single-threaded CPU-intensive (STCI) application and `FFTW` is the partially sequential memory-intensive (PSMI) application. The numbers in parentheses next to the application class indicate the number of threads chosen for that application: the first number for the 1FC-12SC configuration and the second number for the 4FC-12SC configuration.

At a first glance, it can be observed from the workloads that not all possible pairs of classes are actually covered. For the sake of analyzing benchmark pairings that expose diversity in instruction-level and thread-level parallelism we did not pick pairs consisting of co-runners of the same class. Note also that highly parallel memory-intensive benchmarks have been deliberately discarded from these workloads. In preliminary experiments we observed that for benchmark pairings with a highly parallel application (either HPCI or HPMI), schedulers that rely on the number of threads when making scheduling decisions (CAMP and PA) mapped all threads of the HP application on slow cores. The actual reason behind this behavior is that a high number of active threads (this happens most of the time for HP applications) dominates the value of the utility factor and, as a result CAMP and PA schedulers always assign a LOW utility class for all threads, regardless of their memory-intensity (*SF*). For that reason, we only included wupwise_m as a representative HP application (a CPU-intensive parallel benchmark from SPEC OpenMP 2001), discarding other memory-intensive applications of the same suite (such as equake_m or swim_m).

Categories	Benchmarks
STCI-PSMI	`gamess`, `FFTW` (12,15)
STCI-PSCI	`gamess`, `BLAST` (12,15)
STCI-HP	`gamess`, `wupwise_m` (12,15)
STMI-PSMI	`mcf`, `FFTW` (12,15)
STMI-PSCI	`mcf`, `BLAST` (12,15)
STMI-HP	`mcf`, `wupwise_m` (12,15)
PSMI-PSCI	`FFTW` (6,8), `BLAST` (7,8)
PSMI-HP	`FFTW` (6,8), `wupwise_m` (7,8)
PSCI-HP	`BLAST` (6,8), `wupwise_m` (7,8)

Table 2. Multi-application workloads with both single-threaded and multi-threaded applications

For the sake of completeness, we have also studied additional multi-application workloads that combine parallel- and single- threaded applications, but exhibit a wider variety of memory-intensity than those in Table 2, which focus on exploring the impact of thread level parallelism. Table 3 shows this additional set.

Both OpenMP and POSIX threaded applications use adaptive synchronization modes, as such, sequential phases are exposed to the operating system in both cases. Nevertheless, applications implemented using POSIX threads (BLAST, FFT-W) spin for shorter periods of time before blocking (these are the default parameters used in OpenSolaris).

5.3 Aggregate results and detailed analysis of multi-threaded workloads

Figure 4 shows the geometric mean of the speedups achieved by the three asymmetry-aware schedulers (SFD, PA and CAMP) normalized to RR, when running on the AMD platform. Only CAMP is able to deliver performance gains across the wide variety of workloads analyzed in our study, which is the major contribution of this research.

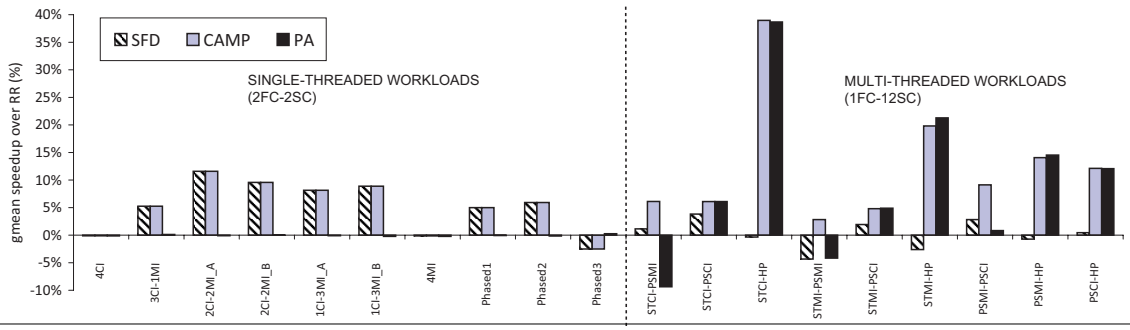

Figure 4. Gmean speedup of SFD, PA and CAMP schedulers when running single threaded and multi-threaded workloads on the AMD platform.

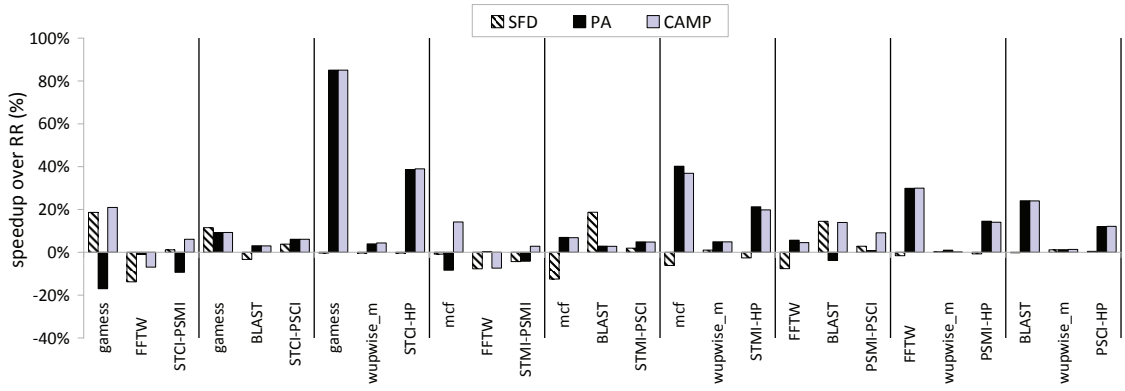

Figure 5. Speedup of asymmetry-aware schedulers on the 1FC-12SC AMD platform.

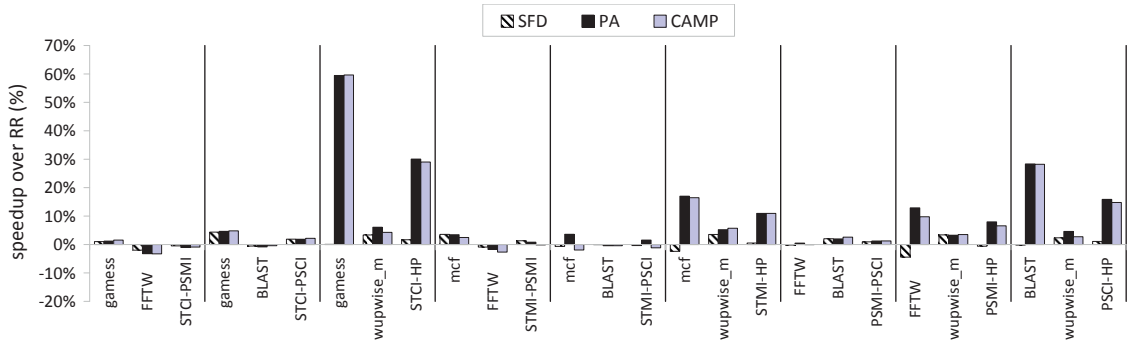

Figure 6. Speedup of asymmetry-aware schedulers on the 4FC-12SC AMD platform.

Categories	Benchmarks
4CI-1PSMI-1HP	`gobmk, h264ref, gamess, povray,FFTW(6),wupwise_m(6)`
4CI-4MI-1HP	`games, gobmk, h264ref, gromacs, milc, mcf, soplex, libquantum, equake_m(8)`
4CI-4MI-1PSMI	`calculix, hmmer, gamess, sjeng, milc, mcf, soplex, libquantum, FFTW(8)`
3CI-1MI-1PSCI	`gamess, gobmk, hmmer, soplex, semphy(12)`

Table 3. Additional multi-application workloads with both single-threaded and multi-threaded applications

Before discussing in detail per-application results, it's worth to analyze the behavior of the partially sequential applications included in the workloads: BLAST (PSCI) and FFTW (PSMI). As opposed to other parallel applications that create all threads at the beginning of the execution, both BLAST and FFTW exhibit several distinct parallel phases where threads are destroyed at the end of a phase and new threads are created at the beginning of the subsequent one. When scheduled by algorithms relying on on-line *SF* monitoring (CAMP and SFD), new spawned threads will have to go through the initial `warm_up` period until they're eligible to be scheduled on fast cores. This means that frequent

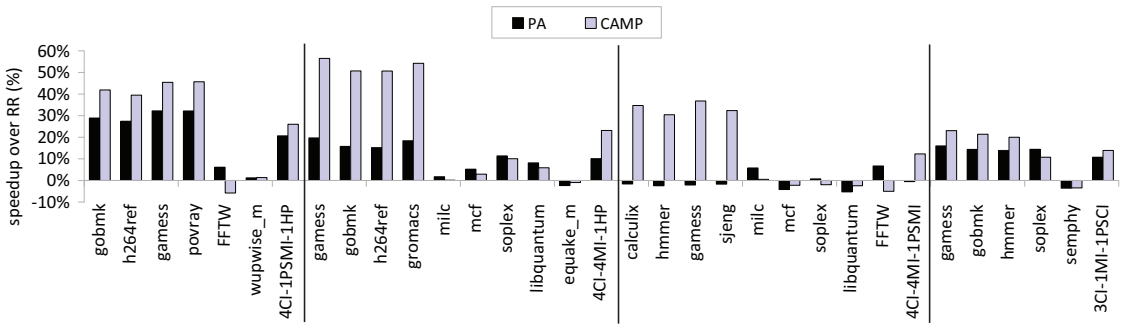

Figure 7. Speedup of PA and CAMP schedulers for additional workloads on the 4FC-12SC AMD platform.

thread creation and destruction might imply that threads will be running on slow cores more often. Common characteristics of both FFTW and BLAST are having significant serial bottlenecks (over 40% of total execution time) as well as CPU-intensive parallel phases.

Serial phases in FFTW (memory-intensive) comprise roughly 80% of the total execution time so we can globally categorize this application as memory-intensive. By analyzing per-thread behavior over time using performance monitoring counters, we found that FFTW's serial phases are, in turn, divided into a very short CPU-intensive phase (at the beginning) and a long memory-intensive phase. According to the boosting feature incorporated into CAMP, the thread executing a sequential phase is initially assigned to the SEQUENTIAL_PART class, since the thread starts exhibiting a CPU-intensive behavior. Later on, when the serial thread enters the memory-intensive phase, CAMP downgrades it into the MEDIUM class.

PA, as well as CAMP supports explicit *boosting* of the priority for running on a fast core for a thread executing a sequential phase of the application. After exploring the effect of varying the customizable parameter `amp_boost_ticks`, we set it to one hundred timeslices (1 second), which ensures the acceleration of sequential phases without monopolizing the fast core.

Figures 5 and 6 show the results for the 1FC-12SC and 4FC-12SC configurations respectively. There is a speedup bar for each application in the workload as well as the mean speedup for the workload as a whole labeled with the name of the workload from Table 2. The first thing to highlight is that RR behaves well when there just a few threads running in the system since all of them will get a significant "slice" of fast cores. Workloads with few threads include those with two PS applications (recall that our PS applications have large phases where only a single thread is active) as well as with one ST and one PS application. Now, we analyze each workload separately for the 1FC-12SC configuration:

- (STCI-PSMB) PA boosts the large sequential phase of FFTW (memory-intensive) at the expense of scheduling the CPU-intensive sequential application (gamess) on slow cores. RR, in contrast, shares the fast cores between the sequential phase of FFTW and gamess, behaving better than PA as a result. CAMP only schedules FFTW on the fast core during the initial CPU-intensive portion of its sequential phase, leaving the fast core available for gamess most of the time. Since gamess is CPU-intensive this is the right way to schedule, and so CAMP beats both RR and PA. SFD primarily runs gamess on the fast core, failing to accelerate the sequential phase of FFT-W.
- (STCI-PSCI) PA and CAMP behave similarly here, because BLAST's sequential phase is also CPU-intensive, so both PA and CAMP schedule it on a fast core. In contrast, RR still schedules BLAST threads on the fast core when it is executing a parallel phase (many active threads), reducing gamess' time on the fast core. Surprisingly, SFD schedules gamess on the fast cores more often than RR does. The reason behind that is that, as stated previously, BLAST creates and destroys threads several times and as a result new spawned threads are not eligible to be scheduled on fast cores until they spend enough time in the warm up period. During this period, gamess is the only CPU-intensive application eligible to run on fast cores.
- (STCI-HP) In this scenario, many CPU-intensive threads are active throughout the execution. RR and SFD perform similarly as a result of fair-sharing the fast core among all threads. On the other hand, PA and CAMP will schedule the single-threaded application in the HIGH utility class (gamess) on the fast core all the time, leaving slow cores for wupwise_m's LOW utility threads. For this reason, CAMP and PA perform significantly better than RR.
- (STMI-PSMI) CAMP does not have many opportunities to improve performance relative to RR. Both mcf and FFTW are primarily memory-intensive, and RR shares the fast core among them. CAMP beats RR by a small amount, only because it schedules FFTW on the fast core during the CPU-intensive portion of its sequential phase. PA primarily schedules FFTW on the fast core due to its large sequential phase, which PA is configured to maximally accelerate. As a result, mcf, an application

with a slightly greater *SF* than the memory-intensive part of FFTW, runs mostly on the slow core.

- (STMI-PSCI) CAMP and PA, which perform similarly here, will schedule the single-threaded mcf on the fast core as long as BLAST is running a parallel phase. When BLAST enters a sequential (CPU-intensive) phase, its active thread will be executed on the fast core, pushing mcf to the slow core. SFD, however, runs the memory-intensive mcf on a slow core while running BLAST's threads on both fast and slow cores, since those threads have a CPU-intensive nature and thus a high speedup factor.
- (STMI-HP) This workload is similar to STCI-HP, since most threads are active for the duration of the experiment. PA schedules the single-threaded application on the fast core and so does CAMP; therefore, they perform similarly. In contrast, SFD will actually schedule mcf on the slow core, since this is the only memory-intensive thread in the workload.
- (PSMI-PSCI) The performance between PA and CAMP in this scenario is dominated by the fact that FFTW's sequential phases are on average much longer than BLAST's. Under the PA scheduler, the first thread executing an application's sequential phase will be placed on the fast core and will not be migrated from it until `amp_boost_ticks` expire or the thread blocks. Long sequential phases of FFTW thus monopolize the fast core and BLAST's sequential phases have little chance to run there, since PA does not share the fast cores equally among threads in the SEQUENTIAL_BOOSTED class. As a result, PA is not exploiting the greater efficiency of BLAST in using fast cores, instead scheduling on fast cores FFTW's memory-intensive sequential phases. CAMP, however, is able to detect memory-intensity in FFTW's sequential phases, downgrading the thread executing it into the MEDIUM class.
- (PSMI-HP) Sequential phases of the PS applications are effectively accelerated by PA and CAMP on the fast core. SFD, on the other hand, is not able to deliver any performance gains, because it schedules the memory-intensive sequential phases of FFTW on slow cores, running on the fast core CPU-intensive threads of parallel (wupwise_m), which gains little speedup when only one of its threads is accelerated.
- (PSCI-HP) As in the PSMB-HP workload, the thread executing sequential phases of the PS application is migrated to the fast core by PA and CAMP. SFD, in contrast will share the fast cores among all threads, since they are CPU-intensive, and as a result it behaves as RR.

In Figure 5 CAMP and PA performed comparably in most cases, because they both considered TLP. CAMP only outperforms PA on the 1FC-12SC when a single-threaded application and a memory-intensive serial thread compete for a fast core. However, on the 4FC-12SC, for the workloads in Table 2 (same benchmarks, different number of threads), PA and CAMP always perform similarly since both schedulers have enough fast cores to effectively accelerate single-threaded applications as well as serial threads (Figure 6). Therefore, there still remains a question, if considering the speedup factor *in addition to* TLP is important for multi-threaded workloads, and in what cases it can bring significant performance improvement over an algorithm that relies on TLP only. Results in Figure 7 answer this final question showing additional workloads with a wider diversity in memory-intensity. In these cases, CAMP does deliver greater performance gains over PA (up to 13%) and demonstrate that considering the speedup factor in addition to TLP brings higher performance improvements.

6. Conclusions

We have presented a comprehensive scheduling algorithm for asymmetric multicore processors. Although the advantages of exploiting efficiency and TLP parallelism on AMPs were well understood before, no one had addressed the design of the corresponding *unified* support in the operating system and evaluated its benefits and drawbacks. Previous asymmetry-aware schedulers employed only one type of specialization (either efficiency of TLP), but not both. As a result, they were effective only for limited workload scenarios. Through our evaluation of a real OS implementation on real hardware we determined that the CAMP scheduler can be effective for a wide variety of applications without requiring their modification. SFD is unable to deliver performance comparable to CAMP for workloads that include multi-threaded applications, while PA is unable to compete with CAMP when applications exhibit a wide variety of memory-intensity. Our overarching conclusion is that in terms of potential for improving performance of software, AMP systems are a viable future alternative to symmetric systems. An essential element for the success of CAMP is a new light-weight technique for discovering which threads utilize fast cores most efficiently.

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.

References

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI Throttling. In *Proc. of ISCA '05*,

pages 298–309, 2005.

[2] 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.

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

[4] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto. Maximizing Power Efficiency with Asymmetric Multicore Systems. *Commun. ACM*, 52(12):48–57, 2009.

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

[6] D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proc. of Eurosys '10*, 2010.

[7] 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*.

[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] V. Kumar and A. Fedorova. Towards Better Performance Per Watt in Virtual Environments on Asymmetric Single-ISA Multi-core Systems. *ACM OSR*, 43(3), 2009.

[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*.

[11] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*, 28(3):26–41, 2008.

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

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