Leveraging Core Specialization via OS Scheduling to Improve Performance on Asymmetric Multicore Systems

JUAN CARLOS SAEZ, Complutense University of Madrid
ALEXANDRA FEDOROVA, Simon Fraser University
DAVID KOUFATY, Intel Labs
MANUEL PRIETO, Complutense University of Madrid

Asymmetric multicore processors (AMPs) consist of cores with the same ISA (instruction-set architecture), but different microarchitectural features, speed, and power consumption. Because cores with more complex features and higher speed typically use more area and consume more energy relative to simpler and slower cores, we must use these cores for running applications that experience significant performance improvements from using those features. Having cores of different types in a single system allows optimizing the performance/energy trade-off. To deliver this potential to unmodified applications, the OS scheduler must map threads to cores in consideration of the properties of both. Our work describes a *Comprehensive* scheduler for *Asymmetric Multicore Processors* (CAMP) that addresses shortcomings of previous asymmetry-aware schedulers. First, previous schedulers catered to only one kind of workload properties that are crucial for scheduling on AMPs; either *efficiency* or *thread-level parallelism* (TLP), but not both. CAMP overcomes this limitation showing how using both efficiency and TLP in synergy in a single scheduling algorithm can improve performance. Second, most existing schedulers relying on models for estimating how much faster a thread executes on a "fast" vs. "slow" core (i.e., the *speedup factor*) were specifically designed for AMP systems where cores differ only in clock frequency. However, more realistic AMP systems include cores that differ more significantly in their features. To demonstrate the effectiveness of CAMP on more realistic scenarios, we augmented the CAMP scheduler with a model that predicts the speedup factor on a real AMP prototype that closely matches future asymmetric systems.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*

General Terms: Algorithms, Performance, Measurement

Additional Key Words and Phrases: Asymmetric multicore, operating systems, scheduling

ACM Reference Format:
Saez, J. C., Fedorova, A., Koufaty, D., and Prieto, M. 2012. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. ACM Trans. Comput. Syst. 30, 2, Article 6 (April 2012), 38 pages.
DOI = 10.1145/2166879.2166880 http://doi.acm.org/10.1145/2166879.2166880

This research was funded by the Spanish government's research contracts TIN2008-005089 and the Ingenio 2010 Consolider ESP00C-07-20811, by the HIPEAC2 European Network of Excellence and by the National Science and Engineering Research Council of Canada (NSERC) under the Strategic Project Grant program.
Authors' addresses: J. C. Saez and M. Prieto, ArTeCS Group, Complutense University of Madrid, Spain; email: {jcsaezal, mpmatias}@pdi.ucm.es; A. Fedorova, SyNAR Group, Simon Fraser University, Burnaby, B.C., Canada; email: fedorova@cs.sfu.ca; D. Koufaty, Intel Labs; email: david.a.koufaty@intel.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 0734-2071/2012/04-ART6 $10.00
DOI 10.1145/2166879.2166880 http://doi.acm.org/10.1145/2166879.2166880

1. INTRODUCTION

Asymmetric multicore processors (AMP) [Kumar et al. 2003, 2004] were proposed as a more power-efficient alternative to conventional multicore processors that consist of identical cores. A performance-asymmetric multicore processor contains at least two core types, "fast" and "slow," which support the same instruction-set architecture, but differ in microarchitectural features, size, power consumption and performance. Fast cores operate at higher clock speeds and may implement sophisticated microarchitectural features to deliver high single-threaded performance. Slow cores operate at lower clock speeds and may have a simpler pipeline, thus occupying a smaller area and consuming less power than fast cores. The resulting savings in area and power of these cores allow higher integration to achieve power-efficient execution of applications with thread-level parallelism (TLP).

Early studies have demonstrated that having just two core types is sufficient to extract most of the benefits from AMPs [Kumar et al. 2003] and this also simplifies their design. Nevertheless, we expect future AMP designs to exhibit a wide variety of fast–to–slow core performance ratios, possibly targeting different market segments [Li et al. 2010].

An AMP can potentially deliver a higher performance per watt than a CMP (Chip Multi-Processor) consisting of identical cores [Hill and Marty 2008; Kumar et al. 2003]. Recent research highlighted that this potential can be realizable by matching each instruction stream with the type of core best suited to this stream [Becchi and Crowley 2006; Kumar et al. 2004]. We refer to this notion of preferring certain types of cores for certain types of computation as *specialization*. The two most common types of specialization are *efficiency specialization* and *TLP specialization*.

Efficiency specialization. Some programs are able to use complex and powerful "fast" cores efficiently. These programs typically have high instruction-level parallelism (ILP), which enables them to effectively use super-scalar out-of-order execution engines typical of fast cores. Other programs, such as memory-bound applications, benefit little from this complex hardware. Efficiency specialization is about mapping to fast cores predominantly instruction streams that use those cores efficiently, which experience performance improvements running on these cores relative to slow cores.

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 energy efficient to run highly parallel code on a large number of small and low-power cores than on a smaller number of fast and powerful cores comparable in area and power [Annavaram et al. 2005]. 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 usually performs poorly on simple slow cores. AMP systems offer the best of both worlds: fast cores can be used to accelerate sequential code, while slow cores enable power-efficient execution of parallel code, thus optimizing performance/watt for code of both types.

To actually benefit from specialization, the system must be equipped with a thread scheduler that maps threads to cores according to the properties of both. Two kinds of operating system schedulers emerged to address this challenge. The first type targeted efficiency specialization [Becchi and Crowley 2006; Kumar et al. 2004; Shelepov et al. 2009]; the second type targeted TLP specialization, by mapping application sequential phases to fast cores and parallel phases to slow cores [Saez et al. 2010b]. Both types of schedulers showed real performance benefits, but the problem is that they each addressed an isolated group of workloads. In this work, we present a *comprehensive*

algorithm CAMP that combines these two specialization types in a single algorithm. Unlike earlier work, CAMP addresses both multithreaded and single-threaded workloads, and in some cases gives better performance than any algorithm that delivers only a single type of specialization.

The biggest challenge in making this possible is to equip the CAMP scheduler with an effective mechanism for deciding which threads are more "profitable" candidates for running on fast cores. To this end, the scheduler must determine the relative benefit that an application would experience when running on fast cores relative to running on slow ones. For single-threaded applications, this benefit can be determined by approximating the *speedup factor* (SF); that is, how much quicker an application retires instructions on a fast core relative to a slow core. In order to estimate this relative benefit for multithreaded applications, we introduce the new metric *Utility Factor* (UF), which accounts for both the speedup factor of the application's threads and its TLP 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 enables the CAMP scheduler to perform effective thread-to-core assignments for multiapplication workloads including both single- and multithreaded applications.

Threads' speedup factors (SFs) can be obtained via *measurement* or *modeling*. Measurement requires running each thread on each core type. Modeling relies on measuring a thread's characteristics on *any* core type and deriving an estimate of the speedup based on a model. The measurement method can severely degrade performance, especially on large multicore systems [Shelepov et al. 2009], and so our CAMP algorithm like several others [Koufaty et al. 2010; Saez et al. 2010a, 2011] uses modeling.

Modeling the speedup factor, however, poses significant challenges since estimation models are inherently architecture specific and asymmetric systems may come in different forms. Performance asymmetry may be based simply on clock frequency (different cores run at different clock speeds but are identical otherwise) or on more substantial differences in microarchitecture, where cores are actually built with different features. While the first type of asymmetry may be used on existing systems in order, for instance, to comply with a target power budget, the second type asymmetry will be seen in future asymmetric systems, such as the recently announced ARM big.LITTLE processor combining Cortex A7 and A15 cores [ARM 2011]. Both types of asymmetry are, therefore, important to address, but previous schedulers were predominantly designed for the first time of asymmetry. In this work we present a model for estimating the speedup factor on a system where cores differ in frequencies as well as on a system where cores have different microarchitectural features (using a prototype asymmetric system from Intel [Koufaty et al. 2010]). To the best of our knowledge, this work is the first to develop a robust SF estimation model for an AMP where cores differ in microarchitectural features.

We implemented CAMP in the OpenSolaris operating system and evaluated it on real multicore hardware. To assess its effectiveness, we used two asymmetric multicore platforms. In the first platform, cores differ in their frequency; in the second one, cores differ in microarchitecture. We demonstrate that CAMP improves performance on both platforms.

We compare CAMP with several other asymmetry-aware schedulers including the Parallelism-Aware (PA) scheduler [Saez et al. 2010b], which performs only TLP specialization; the Speedup-Factor Driven (SFD) scheduler, which only caters to efficiency specialization; and a baseline round-robin (RR) scheduler that simply shares fast cores equally among all threads [Balakrishnan et al. 2005]. For workloads consisting exclusively of single-threaded applications, SFD is sufficient, but this algorithm is ineffective for workloads containing parallel applications. Conversely, PA 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 found that there is extra benefit in using information on efficiency in addition to TLP for realistic workloads containing parallel applications. The greatest benefit of CAMP, therefore, is that it improves performance 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 article is organized as follows. Section 2 explores different estimation models to approximate threads' speedup factors and analyzes their effectiveness. Section 3 describes the utility factor's derivation process. Section 4 presents the design of CAMP and briefly describes other algorithms that we use for comparison. Section 5 shows our experimental results. Section 6 discusses related work. Finally, Section 7 outlines our main findings.

2. ESTIMATING SPEEDUP FACTORS

Most asymmetry-aware schedulers proposed by the research community were evaluated using workloads consisting of single-threaded applications only [Becchi and Crowley 2006; Koufaty et al. 2010; Kumar et al. 2004; Shelepov et al. 2009]. The biggest challenge in designing a scheduler for this kind of workloads is to determine to what extent the performance of individual threads will improve when running on a fast core relative to a slow core. Single-threaded performance is usually measured in terms of instructions per second (IPS), and the relative improvement is referred to as the *speedup factor*. More formally, we define the speedup factor (SF) for a thread 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.

Relying on the speedup factor is sufficient to effectively guide scheduling decisions when only single-threaded applications are running on the system because the SF of an individual thread exclusively determines the speedup that it derives from running on a fast core relative to a slow one. However, the speedup that a multithreaded application experiences when it uses all fast cores in the AMP (relative to using slow cores only) depends not only on the SF of all of its threads, but also on additional factors such as its amount of thread-level parallelism (TLP). As we will see in Section 3, the utility factor (UF) enables us to approximate this speedup for multithreaded applications.

Several online techniques have been proposed to determine speedup factors (SFs). Overall, these can be grouped into two broad categories: those employing direct measurement [Becchi and Crowley 2006; Kumar et al. 2004] and those relying on estimation models [Koufaty et al. 2010; Saez et al. 2011].

The former approach entails running each thread on fast and slow cores to monitor its IPS on both core types and the SF is then computed as the ratio of IPS counts.In previous work, we showed that running each thread on both core types to measure SFs causes serious performance issues, making it inappropriate in practice [Shelepov et al. 2009]; we will elaborate on these issues in Section 6. Since this approach relies on direct measurement of the SF, we refer to it as the *direct measurement* approach.

The latter approach, referred to as the *modeling* approach, relies on estimating a thread's SF using its runtime properties collected on *any* core type, by using, for example, performance monitoring counters during the execution. In this work we opted to follow this approach since it does not require running each thread on both core types and thus overcomes the main shortcomings of direct measurement [Saez et al. 2011].

In the remainder of this section, we explore the design of SF estimation models under two different forms of performance asymmetry: one is due to the processor frequency and another due to differences in cores' microarchitectural features. In the

first scenario, analyzed in Section 2.1, the degree of *memory intensity* of an application enables to approximate reasonably well how much the application will slow down when running on a slow core with respect to a fast core. In the second scenario, covered in Section 2.2, multiple performance limiting factors must be taken into account when estimating the SF, and as a result, designing effective estimation models in this scenario involves additional complexity.

2.1. SF Estimation Model for Cores Differing in Processor Frequency

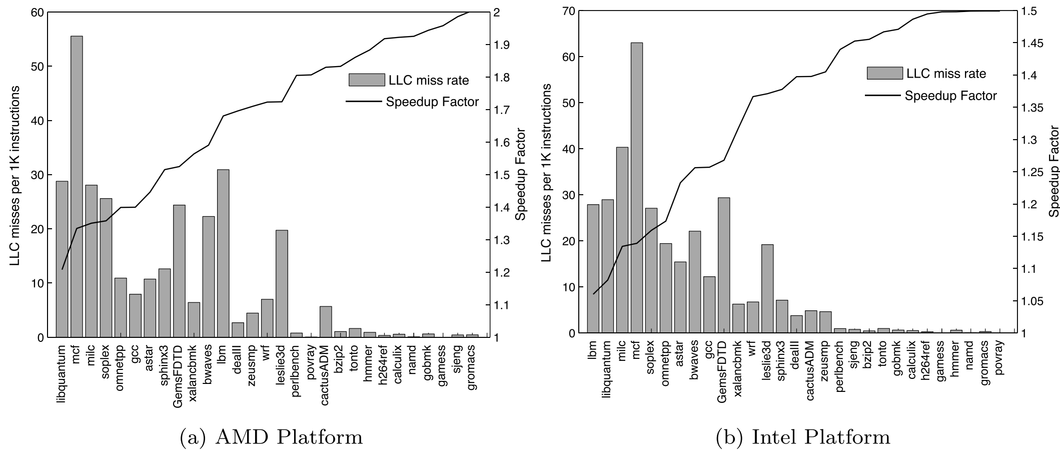
To predict performance variations due to clock frequency, we must consider the application's degree of *memory intensity* [Freeh et al. 2007]. An application with a high rate of memory accesses is likely to stall the core often, so the clock frequency will not have a significant effect on performance. Memory intensity can be approximated by a thread's LLC (last-level-cache) miss rate [Blagodurov et al. 2010].

Our method for computing the speedup factor *SF* relies on threads' LLC miss rates measured online using hardware performance counters. At a high level, the method works as follows. 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 constant number of instructions per cycle (machine dependent) 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 must keep track of the number of LLC misses per 1K instructions retired on the current core type and the memory latency, which can be discovered by the operating system. Because the LLC miss rate does not vary significantly between cores that differ in frequency only, the obtained SF estimates are very similar on both core types. Therefore, we can safely use the same estimation model on both core types.

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 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 nonuniform memory access (NUMA) latencies on some architectures. Accounting for all these factors is difficult, because their complex interrelationship 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 performance results in Section 5.1).

We assessed the effectiveness of our estimation model on two systems made asymmetric by slowing down the frequency of some of the cores. On the first system, which includes AMD quad-core “Barcelona” processors, slow cores operate at the lowest DVFS (dynamic voltage and frequency scaling) level supported by the platform (1.15GHz), while fast cores operate at the maximum DVFS level (2.3Ghz). The second system integrates quad-core Intel Xeon “Clovertown” processors. Its fast cores operate at 3.0 GHz, while slow cores run at 2.0 GHz. We provide further information on our experimental platforms in Section 5.

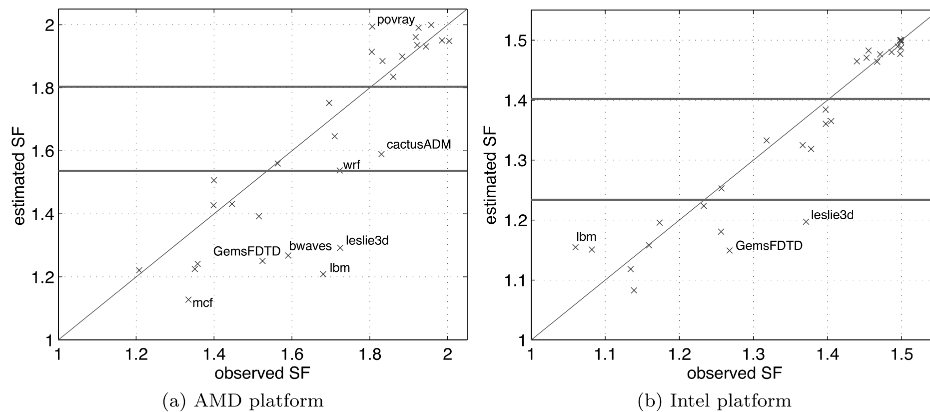
Figures 1(a) and 1(b) illustrate the inverse correlation between the LLC miss rate and the SF for benchmarks in the SPEC CPU2006 suite. We observe that applications with a high miss rate experience low SFs. On the contrary, applications deriving the highest attainable SF on the platform have negligible LLC miss rates. The data also reveal that the inverse correlation between the miss rate and the SF is stronger on the Intel platform than on the AMD platform. We hypothesized that this has to do with the way the LLC miss rate is measured on both platforms. On the Intel platform, LLC misses incurred by the pre-fetching hardware were excluded from the total



(a) AMD Platform

(b) Intel Platform

Fig. 1. Correlation between LLC miss rate and actual speedup factor for all benchmarks in the SPEC CPU2006 suite running on two AMPs where cores differ in processor frequency.



(a) AMD platform

(b) Intel platform

Fig. 2. Observed and predicted speedup factors for all benchmarks in the SPEC CPU2006 suite running on two AMP systems where cores differ in processor frequency. Some outliers have been labeled. Perfectly accurate estimations have all points on the diagonal line. The correlation coefficients for the prediction are 0.86 (AMD) and 0.94 (Intel).

miss count. These cache misses do not usually translate into pipeline stalls, so it is better not to account for them in this context. Unfortunately, the monitoring unit on the AMD system does not filter out prefetching-related misses from the total count, so the obtained value may overcount the miss rate, especially for floating-point benchmarks (such as `leslie3d`, `lbm` or `bwaves`), which lend themselves to prefetching-related optimizations.

Figures 2(a) and 2(b) report how well estimated *SFs* match actual *SFs* for all SPEC CPU2006 benchmarks on the Intel and AMD platforms, respectively. The actual *SF* is measured by running the application on the slow core, then on the fast core, and computing the relative speedup. The estimated *SF* is obtained from the average LLC measured throughout the entire run of the application. As evident, the estimation method is successful in separating highly memory-intensive threads (which are less sensitive to changes in frequency and therefore concentrated towards lower left) from

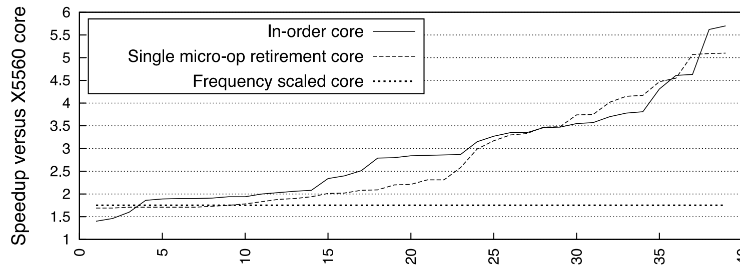


Fig. 3. Speedup of a X5560 core over different slower cores. Each data point represents a different micro-benchmark [Koufaty et al. 2010].

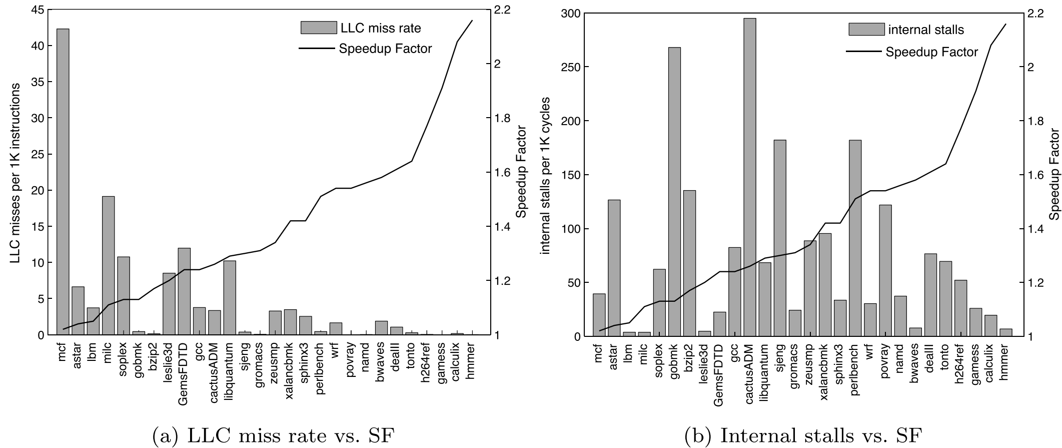
CPU-intensive threads (upper right), but is less precise in classifying mildly memory-intensive benchmarks.

Despite the inaccuracies due to the simplifying assumptions made in our model, we observed that these inaccuracies can be effectively mitigated when it is used in the scheduler. To make this possible, the scheduler classifies applications into three coarse Speedup Factor categories rather than relying on *raw SF* estimates. The boundaries between SF categories are displayed in Figure 2 as horizontal solid lines.

2.2. SF Estimation Model for Cores Differing in Microarchitecture

The emulation of AMPs by downscaling the frequency of some of the cores in symmetric multicore systems has been widely used in both the academic community and the industry to assess the effectiveness of diverse asymmetry-aware scheduling proposals [Annavaram et al. 2005; Li et al. 2007; Saez et al. 2010b; Shelepov et al. 2009]. Nevertheless, Koufaty et al. [2010] demonstrated recently that asymmetric systems where cores differ in frequency only do not exhibit similar performance profiles to those of more realistic systems that integrate cores with different microarchitectures. To illustrate this fact they designed a set of 40 CPU-intensive micro-benchmarks with varying amounts of ILP and measured the wall clock time of each benchmark on an out-of-order core that operates at 2.8Ghz and capable of retiring up to four micro-ops per cycle. They reported the speedup experimented by the benchmarks on this “fast” core with respect to three other slower cores: (1) an in-order core operating at the same frequency as the fast core, (2) a fast core running at the same frequency but retiring up to one micro-op per cycle (single micro-op retirement mode) and (3) a fast core running at 1.6 GHz (frequency scaled). Figure 3 shows the relative performance of these microbenchmarks, displayed in ascending order by ILP. The relative speedup with respect to both an in-order core and a core in single micro-op retirement mode varies with the benchmarks’ ILP. In contrast, the speedup relative to frequency scaling remains the same across the board. As discussed previously, only the memory and cache behavior may affect the relative speedup between cores running at different clock speeds but all of these microbenchmarks are CPU intensive. This result underscores that systems consisting of cores with different frequencies do not model accurately the performance profile of more realistic AMP systems, where cores with different microarchitectural features may be present.

Results in Figure 3 also suggest that using cores with reduced retirement width makes it possible to model the behavior of slower in-order cores more accurately than frequency-scaled cores. This led us [Koufaty et al. 2010] to experimenting with a research AMP prototype consisting of cores that differ in retirement width. In this paper we used a prototype system similar to the one used in Koufaty et al. [2010], which consists of two Intel Xeon E5645 “Westmere” hex-core processors. By using proprietary



(a) LLC miss rate vs. SF (b) Internal stalls vs. SF

Fig. 4. Correlation between different performance metrics and the observed speedup factor for all benchmarks in the SPEC CPU2006 suite experienced on an AMP system where cores differ in microarchitecture.

tools we changed the retirement width in several cores of the system. In particular, slow cores were set to retire up to one micro-op per cycle, whereas fast cores can retire up to four micro-ops per cycle (default setting).

Prior to describing our approach to estimating the speedup factor on the prototype system, we illustrate the main challenges associated with estimating speedup factors under this specific form of performance asymmetry. Figure 4 shows the correlation between different performance metrics and the speedup factor experienced on this system by applications in the SPEC CPU2006 suite. LLC miss rates could be used as a first approximation to determine SFs, but this metric alone is not sufficient to approximate SFs with enough accuracy. As shown in Figure 4(a), benchmarks such as `gobmk`, `sjeng` and `gromacs` exhibit very low LLC miss rates, but experience relatively low SFs. Further factors, in addition to the degree of memory intensity, must be considered when estimating SFs on this platform. In particular, a high number of internal pipeline stalls (as opposed to stalls due to external sources such as memory requests) usually translates into a low SF on this system,[1] which explains why such low SF is experienced by `gobmk` and `sjeng` (as shown in Figure 4(b)). Unfortunately, relying solely on both LLC misses and internal stalls would still incur severe mispredictions. For example, all applications with a low LLC miss rate and a low number of internal stalls would be expected to derive a high SF; that assumption, however, would lead to a major misprediction for `gromacs` (actual SF = 1.3).

The previous discussion highlights that devising an accurate SF estimation model for this form of performance asymmetry can be a rather challenging task for several reasons. First, the metrics capturing the most relevant performance-limiting factors have to be identified by (manually) analyzing correlations between various performance metrics and SFs. Second, the estimation model must combine information about multiple performance metrics.

In the quest of more general and systematic methods to derive SF estimation models, we explored several statistical and machine learning techniques. To that end, we turned to the open source WEKA machine learning package [Hall et al. 2009]. This

[1]Koufaty et al. [2010] found that very frequent internal stalls (such as those associated with branch mispredictions) may lead to substantially reducing the speedup factor on this system even for CPU-intensive applications. These stalls could be approximated by means of *instruction starvation*, a metric accounting for processor internal stalls occurring at the pipeline front-end (Figure 4(b)).

Table I. Selected Metrics

Metric	Description
IPC	Number of instructions retired per cycle
LLC miss rate	Number of L3 (last-level) cache misses per 1K retired instructions
L2 miss rate	Number of L2 cache misses per 1K retired instructions
Execution stalls	Number of cycles no micro-ops are issued per 1K processor cycles
Retirement stalls	Number of cycles no micro-ops are retired per 1K processor cycles

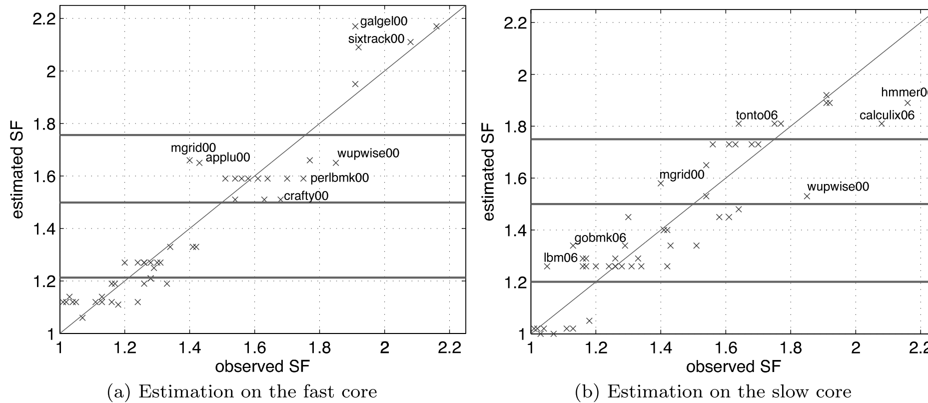
powerful tool is equipped with numerous methods enabling to infer relationships between a set of observations (input attributes) and a target variable. In our setting, the observations correspond to several performance monitoring metrics (collected by means of performance counters) and the target variable is the speedup factor (SF). We found that constructing two separate models (one for predicting the SF from fast-core metrics, and the other from slow-core metrics) provides better accuracy than a unified model. Furthermore, having two separate models makes it easier to cope with *core-specific* metrics such as the number of instructions retired per cycle, whose values on fast and slow cores may differ for the same application.

Among the numerous methods provided by WEKA for numeric prediction, we found that *additive regression* [Friedman 1999] achieved reasonably good accuracy. At a high level, the method takes an initial value for the SF (by convention, the average SF observed by all applications in the training set) and approximates the SF by summing up positive and negative factors to this initial value. Each additive-regression factor is associated with an input metric the factor depends on; input metrics can be associated with zero or more factors. Additive-regression factors are computed sequentially, such that the squared error of the predictions obtained up to that point is minimized.

In order to improve the robustness of the estimation model, we trained[2] the machine-learning method with performance data from all benchmarks in the SPEC CPU2000 and SPEC CPU2006 suites. For those applications, we collected a comprehensive set of performance metrics ranging from general metrics, such as the IPC or LLC miss rate, to more specific metrics approximating stall decomposition on the Intel Westmere processor, such as those related with execution stalls or retirement stalls. Since all the analyzed performance metrics cannot be monitored simultaneously (they exceed the maximum number of performance counters available on the platform), we selected those metrics that contributed more significantly to the resulting SF estimate on each core type (i.e., metrics with the highest additive-regression factors). Selected metrics are shown in Table I. The final SF model for each core type was generated using those metrics as input to the additive-regression engine.

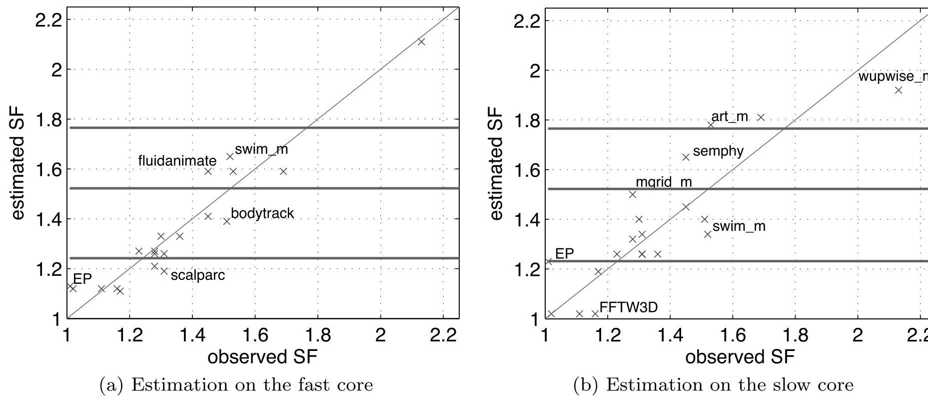
Figures 5(a) and 5(b) show the comparison between estimated and actual *SFs* for benchmarks in the SPEC CPU2006 and CPU2000 suites on fast and slow cores, respectively. At a first glance, we observe that the accuracy achieved on the fast core is significantly better than on the slow one. Achieving a better accuracy on the fast core than on the slow core, however, is not an unexpected result: intuitively, it is much easier to tell which applications would suffer the most from diminishing certain microarchitectural capabilities in the core (fast-to-slow prediction), than identifying which ones would likely experience the highest benefit from adding extra hardware resources (slow-to-fast prediction).

[2]Given that the amount of data for training and testing is limited (data from 54 applications), we employed *cross-validation* to derive the SF models on both core types. As such, each model was repeatedly generated using part of the data for training and the other part for testing.



(a) Estimation on the fast core (b) Estimation on the slow core

Fig. 5. Observed and predicted speedup factors for all benchmarks in SPEC CPU2000 and SPEC CPU2006 running on an AMP system where cores differ in microarchitecture. Some outliers have been labeled. The correlation coefficients for the SF prediction on fast and slow cores are 0.94 and 0.91 respectively.



(a) Estimation on the fast core (b) Estimation on the slow core

Fig. 6. Observed and predicted speedup factors for additional applications running on an AMP system where cores differ in microarchitecture. The correlation coefficients for the SF prediction on fast and slow cores are 0.94 and 0.86 respectively.

In order to complement our assessment of the estimation model designed for this form of performance asymmetry, we experimented with additional applications from different benchmarks suites to those used to generate the model. Figure 6 shows the results for applications in the SPEC OMP2001, NAS Parallel, PARSEC and Minebench benchmark suites, plus a few others. Most applications in these suites are parallel, so, for the sake of simplicity, we collected performance metrics and SFs running all these applications with a single thread. As evident, the model successfully approximates the SF of these applications.

Finally, we must highlight that the scheduler classifies applications into several Speedup Factor categories to mitigate inaccuracies in the estimation, in the same way it does on systems where cores differ in processor frequency. More specifically, the scheduler uses four SF categories to cope with the observed range of SFs (from 1.0 to 2.17).

3. UTILITY FACTOR

The main goal of the CAMP scheduler is to maximize system-wide performance. In other words, CAMP aims to maximize the average IPC (instructions per cycle) of a set of applications running simultaneously on an AMP system. Previous work has highlighted that assigning preferentially to fast cores those applications that experience the highest speedups on these cores (relative to slow ones) contributes to increasing the average IPC of the workload [Becchi and Crowley 2006; Kumar et al. 2004]. To make this possible, the thread scheduler must be equipped with performance models or other heuristics enabling it to estimate applications' relative speedups. In this section, we introduce the Utility Factor (UF), a new metric enabling to approximate relative speedups for both single-threaded and multithreaded applications on AMP systems.

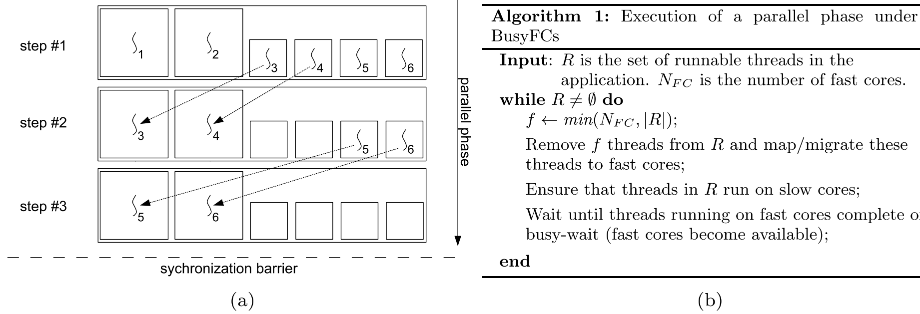
Given a system with N_{FC} fast cores, the *Utility Factor* (*UF*) is a metric approximating the application speedup if *NFC* of its threads are placed on fast cores and any remaining threads are placed on slow cores, relative to placing all its threads on slow cores. For single-threaded applications, this speedup is simply the speedup factor ($UF = SF$). In the remainder of this section, we show how to obtain the UF for multithreaded applications. More specifically, in Section 3.1, we derive a formula for the *UF* using an analytical approach. We observed that using this formula directly in the scheduler implementation gives rise to several problems, so we opted to derive a simpler formula experimentally. Section 3.2 presents this simpler formula that closely approximates the analytically derived version.

3.1. Analytical Derivation for the UF

Asymmetric systems have only a few fast cores. So in multithreaded applications, only a few of the threads will run on the fast cores while the rest are running on slow cores. This makes it challenging to estimate the *UF* for multithreaded applications.

Our goal is to derive a formula enabling the scheduler to estimate how much a multithreaded application would speed up if all fast cores in the AMP were devoted to running threads from this application and assuming that the application runs alone in the system. The speedup is computed with respect to the performance that would result from running all the application threads on slow cores. If the estimated speedup is higher than those of other applications, the application's threads will be assigned preferentially to fast cores; otherwise, threads will be mapped to slow cores. Notably, the same speedup estimate for the application will be used to make these decisions until a change in the active thread count of the application or in the SF of any of its threads takes place. Since the speedup depends on these factors [Saez et al. 2010b], the scheduler will reevaluate the formula to obtain an up-to-date estimate based on the application's current properties.

To make the derivation of the formula tractable, we assume that the application is *perfectly balanced* (i.e., the work is evenly distributed among the threads). This is an assumption that would fit many regular applications. Nevertheless, this assumption is only used to facilitate the derivation of the closed-form analytical model for the *UF* and does not prevent our CAMP scheduler from supporting applications of any kind, including unbalanced applications. These applications usually exhibit phases with limited parallelism where some of their threads do useful work while the remaining ones wait, which results in execution phases with different number of active threads [Saez et al. 2010b]. However, as stated previously, any change in the number of active threads triggers an update operation in the scheduler, so in practice different speedup estimates are used for the various application phases, thus resulting in different scheduling decisions for each phase.



Algorithm 1: Execution of a parallel phase under BusyFCs

Input: R is the set of runnable threads in the application. N_{FC} is the number of fast cores.
while $R \neq \emptyset$ **do**
  $f \leftarrow min(N_{FC}, |R|)$;
  Remove f threads from R and map/migrate these threads to fast cores;
  Ensure that threads in R run on slow cores;
  Wait until threads running on fast cores complete or busy-wait (fast cores become available);
end

(a) (b)

Fig. 7. Execution of a parallel phase under the BusyFCs scheduler on an AMP system.

We further assume that the scheduler always keeps the fast cores busy; we refer to it as *BusyFCs*. Whenever a thread waits on a barrier or exits while running on a fast core, BusyFCs immediately picks another thread running on a slow core and migrates it onto the fast core, thus maximizing its utilization. Note that, in deriving the UF formula, we did not consider the theoretical speedup delivered by a scheduler that fair-shares fast cores among all the application threads. Although such a scheduler may give better performance than using a BusyFCs-like approach in some cases, fair-sharing fast cores entails migrating threads periodically [Saez et al. 2011], which results in a higher number of migrations than those of BusyFCs. These extra migrations may degrade performance significantly, especially for memory-intensive applications, and so we opted to derive the UF formula assuming a BusyFC-like scheduler. Moreover, the behavior of the CAMP scheduler when a highly parallel application runs alone in the system is similar to that of BusyFCs.

In constructing the model, we also assume that the application consists of several parallel phases separated by synchronization barriers. At the beginning of each parallel phase, BusyFCs maps some threads to the N_{FC} fast cores and others to slow cores. Threads running on fast cores will come to the barrier sooner than threads running on a slow core. Once at the barrier, the threads block or spin, effectively making the fast cores idle: the cores will literally go idle in case of blocking or will do useless work in case of spinning. The BusyFCs scheduler will use these freed-up fast cores for some of the threads running on slow cores. Figure 7(a) illustrates this process. Our goal is to model the application performance in this complex scenario relative to the scenario where all threads run on slow cores.

A balanced application consisting of several parallel phases and barriers is one of the most difficult to model and serves as a good base to address other modes of execution. For instance, an application where threads work independently and never synchronize can be approximated as a single parallel phase in our model. An application where synchronization occurs between groups of threads can be modeled as multiple applications with the parallel-phase-then-barrier behavior. We assume this base model in our work only to approximate the UF, our scheduler supports any kind of applications.

We also make three additional simplifying assumptions in constructing the model.

— The number of threads in the application does not exceed the number of cores in the system. This assumption is reasonable because the applications targeted by our scheduler (high-performance CPU-bound parallel applications) are not likely to be run with more threads than cores [van der Pas 2005].
— In many parallel applications, all their threads exhibit very similar speedup factors since they execute the same code with different data. As a result, we opted to

use the *average* speedup factor of all the threads application (denoted by SF_{avg}) to approximate the SF of *any* thread in it.
— The theoretical speedup derived in this section does not account for migration overheads. As a result, the formula approximates an upper bound of the achievable speedup. Nevertheless, the CAMP scheduler takes into account migration overheads when making scheduling decisions, as we will show in Section 4.1.3.

Since we have focused on studying high-performance parallel applications, we use the wall clock (or completion) time to assess performance. Under BusyFCs, determining the completion time of the application boils down to estimating the execution time of the slowest thread for each parallel phase. We determine this execution time by approximating the fraction of instructions executed by the slowest thread on fast and slow cores. To understand how these fractions can be obtained, we analyze the behavior of the BusyFCs scheduler during the execution of a parallel phase, where each thread has to complete a given number of instructions to reach the synchronization barrier at the end of the phase. The sequence of actions performed by the BusyFCs scheduler during the execution of a parallel phase is outlined in Algorithm 1. In each iteration of the algorithm, BusyFCs places up to N_{FC} threads on fast cores and waits till these threads block (i.e., they complete all their remaining instructions until reaching the barrier). When threads block on fast cores, these become idle; then, BusyFCs migrates as many threads as possible from slow to fast cores. Because the scheduler may perform up to N_{FC} slow-to-fast-core migrations in each iteration, all threads will reach the synchronization point in $\left\lfloor \frac{N_{threads}-1}{N_{FC}} \right\rfloor + 1$ iterations. Note that the function $\lfloor x \rfloor$ denotes the largest integer not greater than x.

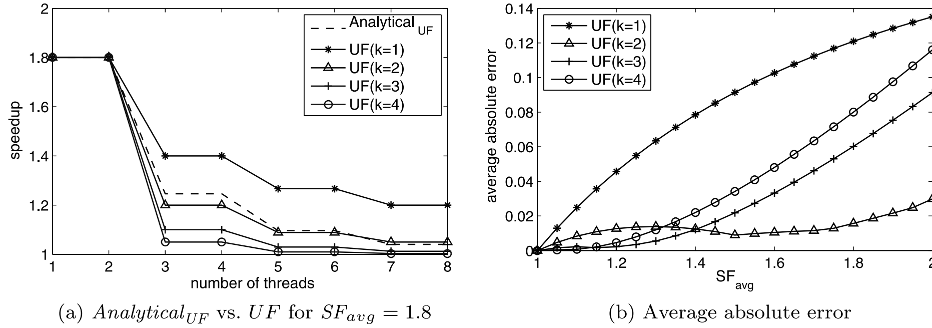
The slowest thread in a given parallel phase is the one mapped to a fast core in the last iteration of Algorithm 1. An estimate of the fraction of instructions executed by this thread on fast and slow cores makes it possible to approximate the completion time of the application under BusyFCs. The detailed derivation process that leads to that estimate as well as to the analytical formula for the UF (speedup under BusyFCs relative to running on slow cores only) can be found in the Appendix. The resulting formula is as follows:

$$Analytical_{UF} = \frac{SF_{avg}}{\left(1 - \frac{1}{SF_{avg}}\right)^{\left\lfloor \frac{N_{threads}-1}{N_{FC}} \right\rfloor} \cdot (1 - SF_{avg}) + SF_{avg}}. \quad (1)$$

3.2. A Simpler Approximation for the UF

Evaluating $Analytical_{UF}$ is a costly operation at kernel level. This has to do with the fact that floating-point operations must be avoided in the kernel (due to the cost associated with saving and restoring the FP registers), so computing the exponentiation in the denominator entails executing a for-loop consisting of integer operations and with a number of iterations equal to the exponent. Note also that the exponent in the denominator is nonconstant and depends on the number of running threads, which can be large on many-core systems. One way to overcome this implementation problem is to precompute the UF values for several commonly occurring SF and $N_{threads}$ values and store them in a lookup table. Another option is to derive a simpler formula. A good approximation is as follows:

$$UF = \frac{SF_{avg} - 1}{\left(\left\lfloor \frac{N_{threads}-1}{N_{FC}} \right\rfloor + 1\right)^{k}} + 1. \quad (2)$$



(a) $Analytical_{UF}$ vs. UF for $SF_{avg} = 1.8$ (b) Average absolute error

Fig. 8. Comparison between the $Analytical_{UF}$ and the UF formulas for different values of k, SF_{avg} and $N_{threads}$.

The variables in the formula are $N_{threads}$ (the number of runnable threads in the application) and SF_{avg} (the average speedup factor of the application's threads). Note that the denominator of this simple formula is also an exponentiation, but the exponent is now a constant k ($k \in \mathbb{N}$). In practice, using $k = 2$ makes it possible to approximate the $Analytical_{UF}$ reasonably well in our experimental platforms, while ensuring less computation cost; evaluating the denominator in the fraction of the UF formula when $k = 2$ entails computing just one integer multiplication regardless of the number of threads in the application. Later on, we will describe how we arrived at this value for k.

Let us describe the intuition behind the UF's formula. The easiest way to understand it is to first consider the case where the application has only a single thread. In this case, $UF = SF_{avg}$; in other words, the utility factor is equal to the speedup that this application would experience from running on a fast core relative to a slow core. Next, let us focus on 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_{avg}. In that case, the denominator is equal to one and $UF = SF_{avg}$. However, if the number of threads is *greater* than the number of fast cores, then, some of threads will be mapped to fast cores for longer periods than others (as explained in Section 3.1) and, as a result, the overall utility factor (speedup) will be less than SF_{avg}. To account for this, we analyzed how the speedup provided by the $Analytical_{UF}$ formula varies with $N_{threads}$ when $N_{threads} > N_{FC}$. We observed that $Analytical_{UF}$ decreases as $\left\lfloor \frac{N_{threads}-1}{N_{FC}} \right\rfloor + 1$ (the number of iterations of the loop in Algorithm 1) increases. Figure 8(a) depicts this trend for a hypothetical balanced application with $SF_{avg} = 1.8$ running with different number of threads on an AMP system consisting of two fast cores and eight slow cores. This observation led us to using the number of iterations of the loop in Algorithm 1 raised to k in the UF formula, hence $\left(\left\lfloor \frac{N_{threads}-1}{N_{FC}} \right\rfloor + 1\right)^k$; the k parameter has been introduced as the exponent to closely track $Analytical_{UF}$. Since the lowest attainable speedup is 1 (no speedup), only $SF_{avg} - 1$ is divided by $\left(\left\lfloor \frac{N_{threads}-1}{N_{FC}} \right\rfloor + 1\right)^k$ in the UF formula.

Figure 8(a) shows a comparison between $Analytical_{UF}$ and UF for different values of k when $SF_{avg} = 1.8$. In this case, choosing $k = 2$ provides the best approximation to the $Analytical_{UF}$. Since this result alone is not enough to guarantee that $k = 2$ will provide good approximations in most cases, we opted to analyze how the absolute error $|Analytical_{UF} - UF|$ varies for different values of k, SF_{avg} and $N_{threads}$. Figure 8(b) shows the results. Error numbers for different values of k are plotted in separate curves. There is one point for each pair (k, SF_{avg}), and each point represents the

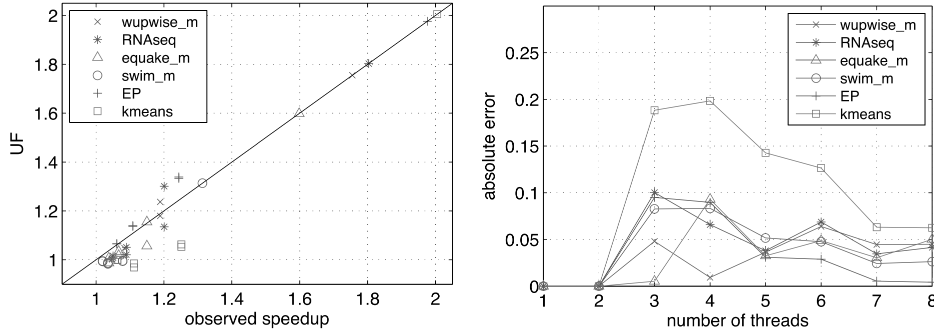


Fig. 9. Comparison between the *UF* and the observed speedup under the BusyFCs scheduler for highly parallel applications running on 2FC-8SC. Both the explicit comparison (left) and the absolute errors (right) are shown. The correlation coefficient is 0.98.

average absolute error obtained for $N_{threads}$ ranging from 1 to 8. Note that we have explored an SF_{avg} range between 1.0 and 2.0 since this is a typical range in our experimental platforms, where fast cores are twice as fast as slow cores at the most (see Section 5). The results reveal that choosing $k = 2$ leads to low absolute errors across the board (the lower the error, the better the estimation), and so this is the actual value we used for our experiments.

Next, we demonstrate experimentally that the UF formula with $k = 2$ closely approximates the actual speedup delivered by the BusyFCs scheduler. Figure 9 shows the comparison between the UF and the actual BusyFCs speedup for several highly parallel applications with different synchronization patterns and memory-intensity levels. These applications belong to the SPEC OMP2001, NAS Parallel and Minebench benchmark suites. Both the observed speedup and the UF have been obtained for $N_{threads}$ ranging from 1 to 8, and so the figures show eight points for each application. To compute the actual performance under BusyFCs, we use our own implementation of this algorithm. The actual speedup was measured on an asymmetric configuration with two fast cores and eight slow cores (2FC-8SC) where cores operate at twice the clock frequency of slow cores. Applications' SF_{avg}s were approximated offline by running the applications with a single thread first on slow cores, then on fast cores and computing the wall clock speedup. As evident, the estimated UF closely tracks the quantity it attempts to approximate. We performed validation for other highly parallel applications from the aforementioned benchmark suites, and found that the results were quantitatively similar (so they are not reported).

Although we only assessed the accuracy of the UF for balanced parallel applications, we must underscore that the UF formula can be also used to estimate the speedup for other types of parallel applications. For example, both unbalanced applications and those including fully sequential phases exhibit execution phases with different speedup [Annavaram et al. 2005; Saez et al. 2010b]. Intuitively, devoting one fast core to accelerate a parallel application when it runs a sequential phase (with a single runnable thread) delivers higher performance benefits that using the fast core to run just one thread of the application when it executes a parallel phase (multiple runnable threads). As stated earlier, the scheduler uses a different speedup estimate (UF value) for application phases with different number of runnable threads (or SFs), thus effectively approximating the speedup for each phase.

At this point, we must highlight that maintaining application-wide UF values, which are shared by all threads of the same application, may limit the scalability of the scheduler. Since the UF formula depends on both SF_{avg} and $N_{threads}$, the application

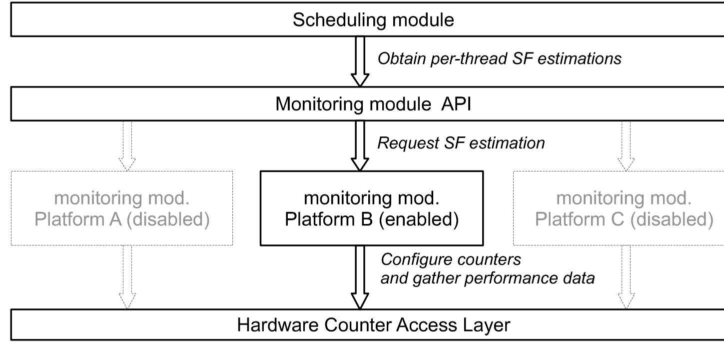


Fig. 10. Components of the CAMP scheduler.

UF must be updated as soon as a change in the runnable thread count or in the SF of any of its threads takes place. In order to avoid contention for accessing the shared UF value from multiple cores, we opted to use per-thread UFs rather than per-application UFs. In order to compute a thread's UF, the CAMP scheduler uses the thread's SF in Equation (2) instead of SF_{avg}. Notably, using per thread UFs leads to similar scheduling decisions than those that would be made when using per-application UFs in most cases. For example, in applications with many active threads the speedup derived from using the scarce fast cores on the AMP becomes negligible regardless of the value of the SF, and so the UF will be similar across threads of the same application ($UF \approx 1$). Furthermore, 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 largely the same, which also leads to similar UF values. For applications with a handful of threads and exhibiting different SF values, those threads with the highest SFs would have more chances of being assigned to fast cores by the scheduler.

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

The CAMP scheduler has a modular design, which is based on two major components: the *scheduling module* and the *monitoring modules*. The *scheduling module* is the code that assigns runnable threads to the different core types and ensures load balance. This component is platform independent; namely, it has been built to work on top of any target AMP system, not assuming any specific form of performance asymmetry. The *monitoring modules*, on the other hand, implement platform-specific SF estimation models, such as those described in Section 2, aimed to provide the scheduling module with up-to-date per-thread SFs as threads go through different program phases.

Figure 10 illustrates the interaction between CAMP's components. The scheduling module makes use of a generic API to periodically obtain per-thread SF estimates provided by the monitoring modules. Several monitoring modules might be available in the system, but only one of them will be *enabled* at a time: the one implementing the SF estimation model for the asymmetric platform in question (e.g., cores with different frequencies). Because SF estimation models rely on monitoring applications' performance during the execution, the monitoring module is in charge of gathering

Table II. Correspondence between UF Values and Utility Classes

Utility Class	**UF range**
HIGH	$UF \geq upper_threshold$
MEDIUM	$medium_threshold \leq UF < upper_threshold$
LOW	$lower_threshold \leq UF < medium_threshold$
VERY_LOW	$UF < lower_threshold$

data from the hardware performance counters. In order to improve portability across different microarchitectures, performance-counter processor-specific operations are not performed directly by the monitoring module, but instead are encapsulated into a hardware-counter access layer.

This modular design makes the CAMP scheduler easily extensible to new forms of performance asymmetry; adding support for a new form of performance asymmetry entails creating the associated platform-specific monitoring module. In our implementation, monitoring modules have been implemented as dynamically loadable kernel modules. CAMP's scheduling module has been implemented as a new Solaris scheduling class that attempts to maximize the average performance of multiapplication workloads running on AMPs. It targets primarily single-threaded and high-performance parallel applications rather than other applications such as web servers, for which achieving high throughput and acceptable response times is more important than improving the overall system performance.

The remainder of this section is organized as follows. Section 4.1.1 describes the CAMP scheduling algorithm. Section 4.1.2 provides further details on the inner workings of CAMP's monitoring modules. Finally, Section 4.1.3 gives an overview of our approach to mitigating the overhead associated with thread migrations in the scheduler implementation.

4.1.1. The Scheduling Module: The CAMP Algorithm. As stated in Section 3, CAMP aims to maximize the performance of a set of applications that run simultaneously on an AMP system. To make this possible, CAMP's scheduling module classifies threads into *utility classes* (based on their UFs) and preferentially assigns to fast cores those threads with the highest UFs. Using classes instead of raw UF values makes it possible to mitigate some inaccuracies in the estimation of a thread's *SF* (used in the UF formula), as well as to provide comparable treatment for threads whose UFs are very close.

According to their utility factors, threads are categorized into four utility classes: VERY_LOW, LOW, MEDIUM and HIGH. Table II shows the utility class that will be assigned to each thread based on its UF value. Utility classes are displayed in the table in descending order by priority to run on fast cores. Hence, threads falling into the HIGH utility class would be preferentially assigned to fast cores, followed by threads in the MEDIUM, LOW and VERY_LOW classes, respectively. This classification relies on three utility thresholds (*lower*, *medium* and *upper*) which determine the boundaries between utility classes. In our experimental evaluation, we have used different threshold values for each platform. Thresholds have been chosen such that the number of utility-class mispredictions for the diverse application sets shown in Sections 2.1 and 2.2 is minimized.

The VERY_LOW utility class is used primarily for threads from applications that run phases with a high amount of TLP. As pointed out in Section 3, highly parallel applications derive speedups as low as 0% from using all fast cores available in the system, and, as a result, threads from those applications exhibit low UF values ($UF \approx 1$). Most sequential applications, on the other hand, experience nonnegligible

speedups on our target asymmetric systems, so, in practice, the HIGH, MEDIUM and LOW utility classes are primarily used to categorize serial code. Finally, mildly parallel applications (i.e., applications running with a handful of threads only) typically fall into the LOW and VERY_LOW utility classes.

At a high level, the assignments of threads to core types are performed as follows. Threads falling in the HIGH utility class will run on fast cores. 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 and very-low-utility threads, respectively (we do not optimize for power consumption in this work, so our scheduler tries to keep fast cores as busy as possible).

If there are more high-utility threads than fast cores, fast cores will be fair-shared among these threads using a round-robin mechanism that was presented in our earlier work [Saez et al. 2011]. This mechanism relies on periodic thread migrations, that, in our setting, take place every 2 seconds on average. As we will show in Section 5, fair-sharing fast cores among high-utility threads gives better average performance that running only some of these threads on fast cores. In practice, fair-sharing is only triggered for single-threaded applications with a high utility factor. As explained below, multithreaded applications will most often fall into the LOW utility class. Therefore, for multithreaded applications, CAMP's behavior is most similar to that of BusyFCs, hence the assumption used in the derivation of the UF.

In contrast with threads in the HIGH utility class, fast cores will not be shared equally for threads in the remaining classes. Sharing the cores equally implies cross-core migrations as threads are moved between fast and slow cores. As shown in our previous work [Saez et al. 2010b], these migrations degrade performance, especially for memory-intensive threads, because threads may lose their last-level cache state as a result of migration.

Threads of parallel applications executing a sequential phase will be designated to a special class: SEQUENTIAL_BOOSTED. These "serial" threads will get the highest priority for running on fast cores: this provides more opportunities to accelerate sequential phases. Only serial threads with $UF \geq upper_threshold$, however, will be assigned to the SEQUENTIAL_BOOSTED class. Threads in other classes will remain in their regular class despite running sequential phases. We opted not to give these threads an elevated status because, first they do not use fast cores efficiently; and second, that may result in relegating high-utility threads to slow cores. Note that when a thread in the SEQUENTIAL_BOOSTED class runs on a fast core, CAMP only fair shares the remaining fast cores among high-utility threads. To prevent the monopolization of fast cores, threads in the SEQUENTIAL_BOOSTED class will be granted this elevated status for `amp_boost_ticks` clock ticks (a configurable parameter) at the most. After that period, they will be downgraded to their regular class, as determined by the utility factor. After exploring the effect of varying the `amp_boost_ticks` parameter, we opted to set it to fifty time slices (0.5 seconds), which ensures the acceleration of sequential phases without monopolizing fast cores.

4.1.2. The Monitoring Module: Determining Speedup Factors. In Section 2, we analyzed different estimation models enabling us to approximate threads' speedup factors on our target platforms. Such estimation models are implemented as platform-specific monitoring modules that feed the scheduling module with up-to-date per-thread speedup factors as these go through different program phases.

The monitoring module uses hardware performance counters to keep track of the necessary performance metrics used as input to the SF estimation model in question. Performance counters are sampled every 20 timer ticks (roughly 200ms on our

experimental system). We observed that the overhead associated with sampling performance counters and estimating speedup factors becomes negligible at this sampling rate.

We keep a running average of the estimated SFs observed at different sampling 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 SF to spike intermittently after migration.

CAMP's monitoring modules implement 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 overheads. Instead, *SF* estimations are updated exclusively once a thread enters a phase of stable behavior. To detect those stable phases, we used a light-weight mechanism based on a `phase_transition_threshold` parameter (12% in our experimental platform). When the running 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.

4.1.3. Topology-Aware Design. An important challenge in implementing any asymmetry-aware scheduler is to reduce 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 do not share the last-level cache (LLC).

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. Apart from making the scheduler aware of applications' sensitivities to cross-core migrations, it is also worth considering ways to make migrations less expensive. In particular, if AMP systems are designed such that there is at least a fast core in each *memory-hierarchy domain* (i.e., sharing a LLC with some slow cores), migration overhead can be mitigated. In 2010b we demonstrated 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, so they are *topology aware*.

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 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. We observed that this variance is especially high when running multiapplication workloads consisting of single-threaded applications only. Nevertheless, as shown in Saez et al. [2011], RR's performance is comparable to or better than the default scheduler.

To the best of our knowledge, the PA scheduler is the first OS-level scheduler delivering TLP specialization [Saez et al. 2010b]. The PA scheduler has the same code

base as CAMP, but since PA accounts only for TLP, the utility factor as such cannot be used to classify threads. For that reason, the *SF* value in the *UF* formula is replaced by a constant representing the upper bound of the achievable *SF* on a given system: the ratio between the maximum IPS attainable on a fast and on a slow core. 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 is not aware of threads' SFs, it cannot distinguish between HIGH, MEDIUM, LOW and VERY_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 also uses the *UF* formula to drive scheduling decisions, but because it does not account for the TLP of the application, the number of threads is always equal to one. The SFD scheduler estimates the *SF* by means of the same estimation models used by CAMP.

The RR algorithm shares fast cores among all threads using the same mechanism that CAMP uses to share fast cores among high-utility threads.

5. EXPERIMENTS

We have assessed the effectiveness of the CAMP scheduler using real multicore hardware made assymetric by reducing either the processor frequency or the retirement width in some of the cores. We used the following multicore server systems.

— *Intel-8* consists of two Intel Xeon X5365 "Clovertown" quad-core processors running at 3.0GHz (8 cores). Each processor integrates two dies with two cores each sharing a 4MB L2 cache. L1 instruction and data caches, with 32KB each, are private to each core.
— *Intel-12* consists of two Intel Xeon E5645 "Westmere" hex-core processors running at 2.4GHz (12 cores). Each processor includes a shared 6MB L3 cache and two other private cache levels: a 256KB L2 unified cache and 64KB L1 instruction and data caches.
— *AMD-16* integrates four AMD Opteron 8356 "Barcelona" quad-core processors running at 2.3GHz (16 cores). Each chip includes a 2MB shared L3 cache and two other private cache levels: a 512KB L2 unified cache and 64KB L1 instruction and data caches.

We experimented with applications from several benchmark suites (SPEC CPU 2006, SPEC OMP 2001, PARSEC, NAS Parallel Benchmarks and Minebench), as well as with `BLAST` (a bioinformatics benchmark) and `FFTW` (a scientific benchmark performing the fast Fourier transform). Both OpenMP and POSIX-threaded applications use adaptive synchronization modes, as such, large sequential phases are exposed to the operating system [Saez et al. 2010b].

We analyzed the behavior of the investigated schedulers under two types of multiapplication workloads: the former consist of single-threaded applications, typically targeted by algorithms like SFD, and the latter includes multithreaded applications, typically targeted by algorithms like PA. 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 [van der Pas 2005].

In all workloads, 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. The average completion time for all the executions of a benchmark under a particular asymmetry-aware scheduler is compared to that under RR, and the wall clock speedup is reported.

The evaluation section is divided into two parts. Section 5.1 focuses on the analysis of the CAMP scheduler on asymmetric systems where cores differ in processor frequency. Section 5.2 reports our results obtained on asymmetric multicore configurations consisting of cores that differ in microarchitecture.

5.1. Results for Asymmetric Systems where Cores Differ in Processor Frequency

In this section, we explore several asymmetric configurations including cores with different frequency. These configurations are based on the Intel-8 and the AMD-16 platforms (described previously). The frequency of fast and slow cores was set to the maximum and minimum frequency (DVFS) levels, respectively. In particular, on Intel-8's asymmetric configurations, fast cores operate at 3.0 GHz, while slow cores run at 2.0 GHz. On the AMD platform, in contrast, higher performance differences between core types are obtained since fast cores operate at twice the frequency of slow cores (at 2.3 GHz and 1.15 GHz, respectively). Because the AMD-16 platform supports core-level DVFS, we are able to vary the frequency for each core independently. On the Intel-8 platform, however, cores in the same physical package (sharing an L2 cache) are within the same power domain, so they must operate at the same voltage/frequency level.

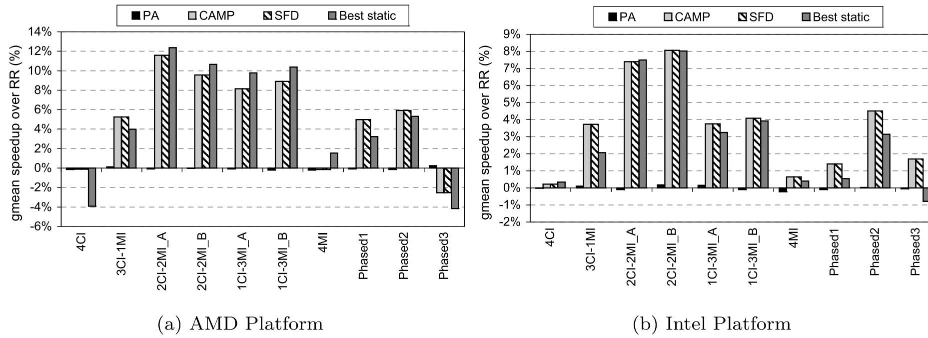
We used three AMP configurations: (1) 1FC-12SC, one fast core and 12 slow cores, the fast core is on its own chip and the other cores on that chip are disabled; (2) 4FC-12SC, four fast cores and 12 slow cores, and (3) 2FC-2SC, two fast cores, two slow cores, none of them sharing a last-level cache with one another. The first two configurations were emulated on AMD-16, while the third one was replicated on both Intel-8 and AMD-16.

5.1.1. Workloads Consisting of Single-Threaded Applications. As stated in Section 2, an application's memory-intensity level determines its sensitivity to variations in the processor frequency. For that reason, workloads evaluated in this section attempt to cover a broad spectrum of memory-intensity levels. To this end, we selected eleven applications from the SPEC CPU 2006 suite and constructed ten workloads containing representative sets. Some benchmarks are primarily memory intensive (such as `mcf` or `milc`); others are primarily CPU intensive (such as `gromacs` or `sjeng`). There is also a *hybrid* application type that exhibits long-term memory-intensive phases interspersed with long-term CPU-intensive phases across their execution. The `astar` benchmark falls into this class.

The selected workloads are displayed in Table III. 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.

Table III. Multiapplication Workloads Consisting of Single-Threaded Applications

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



(a) AMD Platform (b) Intel Platform

Fig. 11. Speedup of PA, SFD, CAMP and Best Static schedulers when running single-threaded workloads on the 2FC-2SC configuration.

Figure 11 shows the results for these workloads. To complement our assessment on the effectiveness of SF predictions, we provide a comparison with a “Best Static” assignment, which ensures applications with the highest overall SFs 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, fair-shares fast cores among them. (Recall that PA assigns all single-threaded applications to the HIGH utility class.) 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 close to “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 estimations, as shown in Section 2.1). Notably, “Best Static” does not always guarantee optimal mappings. This is the case for application sets exhibiting different SF phases and for workloads including a number of high-utility threads greater than the number of fast cores. In the latter case, fair-sharing fast cores among high-utility threads gives better performance on average than the one resulting from assigning some of these threads to fast cores. For that reason, both CAMP and SF deliver comparable or better performance than “Best Static” for the 4CI and 3CI-1MI workloads.

5.1.2. Workloads Consisting of Single-Threaded and Multithreaded Applications. We categorized applications into three groups with respect to their parallelism: highly parallel applications (HP), partially sequential (PS) applications (parallel applications with

Table IV. Multiapplication Workloads with Both Single-Threaded and Multithreaded Applications

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)

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 (as shown in Table IV). As in Table III, the categories in the left column are listed in the same order as the corresponding benchmarks. 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 this workload set 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 this workload set. 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 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 OMP2001).

For the sake of completeness, we have also studied additional multiapplication workloads that combine parallel- and single-threaded applications, but exhibit a wider variety of memory intensity than those in Table IV. Workloads in this additional set (shown in Table V) enable us to explore the combined impact of thread-level and instruction-level parallelism further. Notably, as opposed to the benchmarks pairings shown in Table IV, the additional workload set does include HP memory-intensive (HPMI) applications, such as equake_m.

Table V. Additional Multiapplication Workloads with Both Single-Threaded and Multithreaded Applications

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

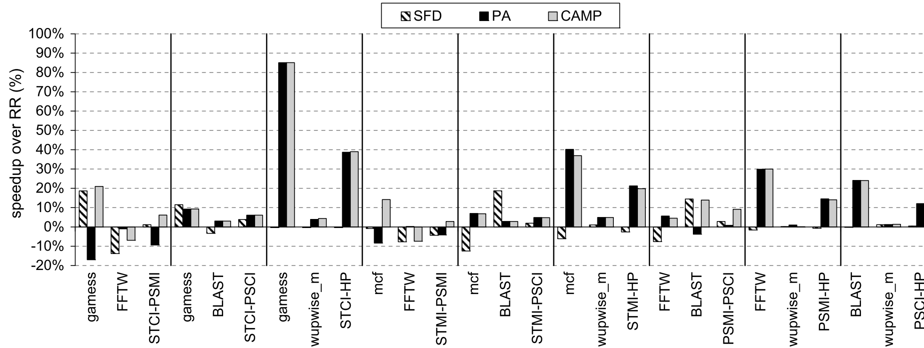


Fig. 12. Speedup of asymmetry-aware schedulers on 1FC-12SC.

Before discussing in detail per-application results, it is worth analyzing 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 dynamically created and destroyed. When scheduled by algorithms relying on online *SF* monitoring (CAMP and SFD), new spawned threads will have to go through the initial warm_up period until they are eligible to be scheduled on fast cores. This means that frequent thread creation and destruction might imply that threads will be running on slow cores more often. Moreover, it is worth noting that both FFTW and BLAST have significant serial bottlenecks (over 40% of total execution time) and CPU-intensive parallel phases.

Serial phases in FFTW (memory intensive) constitute a large fraction 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_BOOSTED 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.

Figure 12 shows the results for the 1FC-12SC configurations. 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 IV. The first aspect to highlight is that RR behaves well when there are just a few threads running in the system since all of them get a significant "share" of fast-core cycles. Workloads with few threads include those with two PS applications (recall that 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-PSMI) 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 FFTW.
— (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's share of fast-core time. Surprisingly, SFD schedules gamess on the fast cores more often than RR does. The reason behind this behavior is that BLAST creates and destroys threads several times and as a result new spawned threads are not eligible to be scheduled on fast cores until their warm-up periods expire. During these periods, 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 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 here. 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, 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 is 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 schedules mcf on the slow core, since this is the only memory-intensive thread in the workload.
— (PSMI-PSCI) The performance differences between PA and CAMP in this scenario are 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 is placed on the fast core and will not be migrated from it until amp_boost_ticks expire or the thread blocks. The long sequential phases of FFTW leads to monopolizing the fast core and BLAST's sequential phases have little chance

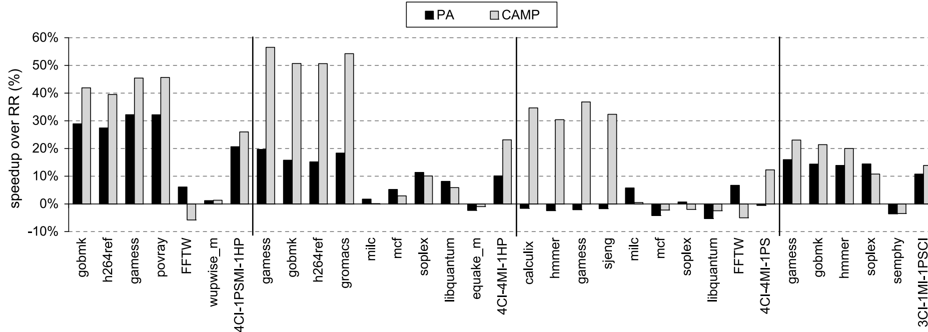


Fig. 13. Speedup of the PA and CAMP schedulers for additional workloads on 4FC-12SC.

to run there, since PA does not share the fast cores equally among threads in the SEQUENTIAL_BOOSTED class. As a result, PA does to cater to the greater efficiency of `BLAST` in using fast cores, and resorts instead to running `FFTW`'s memory-intensive sequential phases on fast cores. CAMP, however, is able to detect `FFTW`'s memory-intensive sequential phases, successfully 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 shares the fast cores among all threads, since they are CPU intensive and then, it behaves as RR.

Results in Figure 12 reveal that CAMP and PA, which consider TLP, performed comparably in most cases. CAMP only outperforms PA on 1FC-12SC when a single-threaded application and a memory-intensive serial thread compete for a fast core. However, for workloads in Table IV running on the 4FC-12SC configuration (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. We do not report these results here due to space limitations.

Therefore, there still remains a question: in what cases does considering the speedup factor *in addition to* TLP bring significant performance improvements over an algorithm that relies on TLP only? Results in Figure 13 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%), which demonstrates that considering the speedup factor in addition to TLP brings higher performance improvements.

5.1.3. Aggregate Results. Figure 14 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-16 platform. Only CAMP is able to deliver performance gains across the wide variety of workloads analyzed, which is the major contribution of this research.

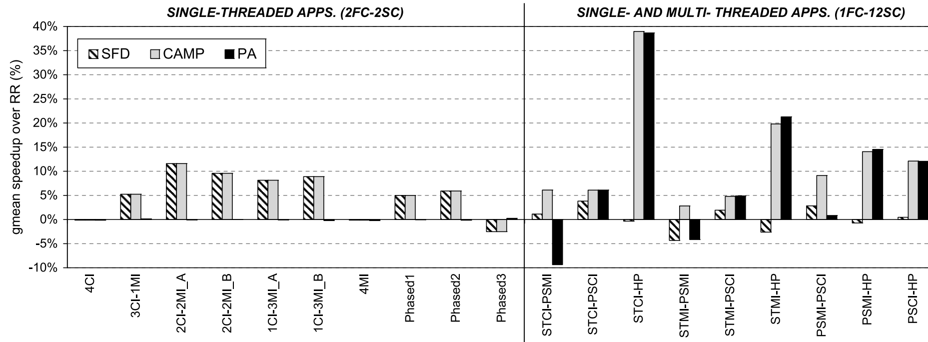


Fig. 14. Gmean speedup of SFD, PA and CAMP schedulers when running single threaded and multithreaded workloads on the AMD-16 platform.

5.2. Results for Asymmetric Systems where Cores Differ in Microarchitecture

In this section we analyze the performance of CAMP, SFD and PA on several AMP configurations based on Intel-12. As stated earlier, we used Intel proprietary tools to introduce asymmetry in the system by reducing the retirement width of several cores (slow) to one micro-op per cycle; the remaining cores (fast) use the default setting, thus retiring up to four micro-ops per cycle. Both fast and slow cores operate at the same frequency (2.4Ghz).

To carry out the evaluation, we used three AMP configurations: (1) 2FC-2SC-A, two fast cores and two slow cores in the same chip, all sharing a last-level cache with one another; (2) 2FC-2SC-B, two fast cores and two slow cores distributed into two chips such that there is a fast and a slow core per chip sharing a last-level cache; and (3) 2FC-10SC, two fast cores and ten slow cores in two chips, each chip includes one fast core and five slow cores. The 2FC-2SC-A and 2FC-2SC-B configurations consist of fewer cores than those included in Intel-12, so unused cores were disabled.

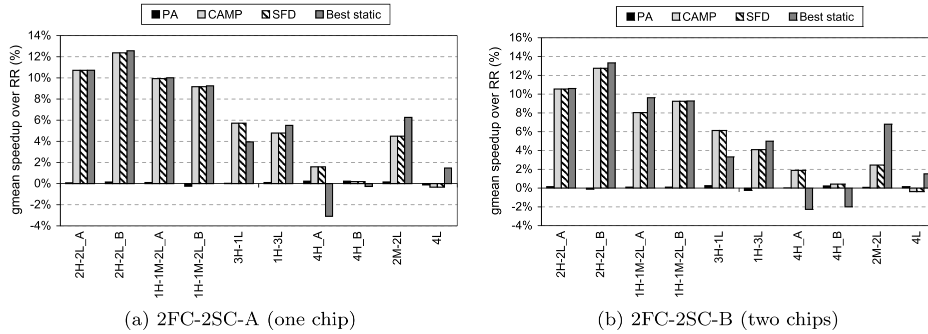
5.2.1. Workloads Consisting of Single-Threaded Applications. An application's degree of memory intensity affects the efficiency that the application derives from running on a core with a high retirement width relative to another capable of retiring fewer instructions. However, several other performance-limiting factors (such as branch mispredictions) may also contribute to the resulting relative speedup (as shown in Section 2).

The workloads evaluated in this section cover a rich set of scenarios. In constructing the workloads, we divided applications in the SPEC CPU2006 suite into three classes based on their speedup factors: high (H), medium (M) and low (L). Table VI shows the ten selected workloads. The central column of the table specifies the workload composition. For example, the 2H-2L_A and 2H-2L_B workloads consist of two high-SF (H) benchmarks and two low-SF (L) benchmarks. Notably, we observed that applications from SPEC CPU2006 exhibiting distinct long-term SF phases over time under cores with different frequency (such as `astar`) present a more uniform SF profile on cores differing in microarchitecture. As a result, this workload set does not include applications that alternate between long-term high-SF phases and long-term low-SF phases (unlike workloads in Table IV).

Workloads in Table VI can be divided into several categories. The first four workloads are *highly heterogeneous*: they show a wide diversity of speedup factors, and so they are likely to experience large performance improvements from asymmetry-aware algorithms exploiting efficiency specialization [Saez et al. 2011]. The second category of workloads is *moderately heterogeneous*, which encompasses the 3H-1L, 1H-3L and

Table VI. Multiapplication Workloads Consisting of Single-Threaded Applications

Categories	**Benchmarks**
2H-2L_A	`calculix`, `gamess`, `gobmk`, `milc`
2H-2L_B	`hmmer`, `calculix`, `soplex`, `astar`
1H-1M-2L_A	`hmmer`, `perlbench`, `soplex`, `mcf`
1H-1M-2L_B	`gamess`, `namd`, `astar`, `gobmk`
3H-1L	`hmmer`,`calculix`, `gamess`, `gobmk`
1H-3L	`gamess`, `sjeng`, `GemsFDTD`, `leslie3d`
2M-2L	`povray`, `gromacs`, `milc`, `mcf`
4H_A	`hmmer`, `hmmer`, `calculix`, `gamess`
4H_B	`calculix`, `calculix`, `gamess`, `gamess`
4L	`gobmk`, `gobmk`, `astar`, `astar`



(a) 2FC-2SC-A (one chip) (b) 2FC-2SC-B (two chips)

Fig. 15. Speedup of PA, SFD, CAMP and Best Static schedulers when running single-threaded workloads on Intel-12.

2M-2L workloads. These workloads include benchmarks with less extreme differences among SFs, and so they are expected to benefit less from asymmetry-aware scheduling than highly heterogeneous workloads. Finally, we explored *lightly heterogeneous* workloads, consisting of applications whose SFs are very close. The 4H_A, 4H_B and 4L workloads fall into this category.

The results for these workloads running under PA, CAMP, SFD and the "Best Static" assignment are shown in Figure 15. As explained earlier, PA resorts to fair-sharing fast cores among threads since it is unaware of SFs, and that leads it to matching the performance of RR (negligible speedup). Recall that the SFD and CAMP schedulers behave similarly in this scenario because $SF = UF$ for single-threaded applications and, at the same time, both schedulers rely on the same SF estimates to drive scheduling decisions.

The results in Figure 15 reveal that benefits from the CAMP and SFD schedulers, which exploit efficiency specialization, are specially pronounced for highly heterogeneous workloads (the first four workloads in the figures). For most of these workloads, CAMP and SFD perform similarly to the oracular Best Static. We also observe that for lightly heterogeneous workloads consisting of three or more high-SF benchmarks (3H-1L, 4H_A and 4H_B), CAMP and SFD outperform Best Static. The main takeaway from this result is that fair-sharing fast cores among high-utility threads under these circumstances does not only help mitigate inaccuracies in the model but it also results in higher aggregated performance than Best Static's, which may map some high-utility threads to slow cores. For the remaining mildly and lightly heterogeneous workloads

Table VII. Multiapplication Workloads Consisting of Multithreaded and Single-Threaded Applications

Categories	**Benchmarks**
2STH-2STL-1HPH	hmmer, calculix, gobmk, astar, wupwise_m (8)
2STH-2STL-1HPL	hmmer, gamess, soplex, bzip2, EP (8)
2STH-1PSL-1HPH	calculix, gamess, BLAST (5), wupwise_m (5)
1STH-1STL-1PSM-1HPM	hmmer, gobmk, semphy (5), fma3d_m (5)
1STH-1STL-1PSM-1PSL	calculix, bzip2, semphy (5), BLAST (5)
1STH-1STL-1PSL-1HPH	hmmer, astar, bodytrack (5), wupwise_m (5)
2STH-1PSM-1PSL	calculix, gamess, bodytrack (5), FFTW (5)
2STM-1PSL-1HPH	wrf, namd, FFTW (5), wupwise_m (5)
3STH-3STL-1PSL	hmmer, hmmer, gamess, gobmk, astar, BLAST (6)

(1H-3L, 2M-2L and 4L), which include at least two low-SF applications, we observed that CAMP and SFD performed different thread assignments to those of Best Static; we found that this is due to inaccuracies in the SF estimation model. Even in those difficult conditions, CAMP and SFD are able to outperform PA and RR.

5.2.2. Workloads Including Both Single- and Multithreaded Applications. For the evaluation in this section, we constructed nine workloads consisting of both sequential and parallel applications covering a broad spectrum of speedup factors. The workloads are shown in Table VII. The categories in the central column of the table provide information on the parallelism class of each benchmark in the workload (ST, PS or HP) along with their speedup factor class (L, M or H). The selected workloads include applications from all possible parallelism/SF classes, but one: partially sequential applications with high speedup factor (PSH class). Unfortunately, we did not find any application falling into this class in the parallel benchmarks suites we explored. In particular, it is worth highlighting that the BLAST benchmark, which derives a high speedup factor on our asymmetric systems where cores differ in processor frequency (as shown in Section 5.1.2), experiences a negligible SF on this system.[3] Therefore, BLAST is labeled as PSL.

Figure 16 shows the results obtained for workloads in Table VII on 2FC-10SC. At a first glance, we can see that SFD scheduler matches the performance of CAMP for some workloads. However, the results also reveal that SFD makes inefficient use of the platform for workloads including HPH (highly parallel, high-SF) applications. In these scenarios, SFD “wastes” fast-core cycles in running high-SF code of threads in HPH applications, while CAMP effectively categorizes these threads as low-utility (highly parallel code) and relegates them to slow cores. As pointed out earlier, the PA scheduler is unable to match the performance of CAMP for workloads including a significant amount of serial code, and at the same time, exhibiting a wide diversity of speedup factors. Since many of these workloads have these characteristics, we observe higher performance differences between both schedulers (CAMP outperforms PA by up to 16%).

5.2.3. Aggregate Results. Figure 17 shows the geometric mean of the relative speedups achieved by the SFD, PA and CAMP, when running on Intel-12–based asymmetric

[3]Basic Local Alignment Search Tool (BLAST) is a program extensively used in bioinformatics research to determine the closest match of a new protein or amino acid sequence in an existing database of known sequences. We observed that this program does not derive any extra benefit from the fast core (with higher retirement width than the slow one) since it incurs a significant number of pipeline stalls due to frequent branch mispredictions.

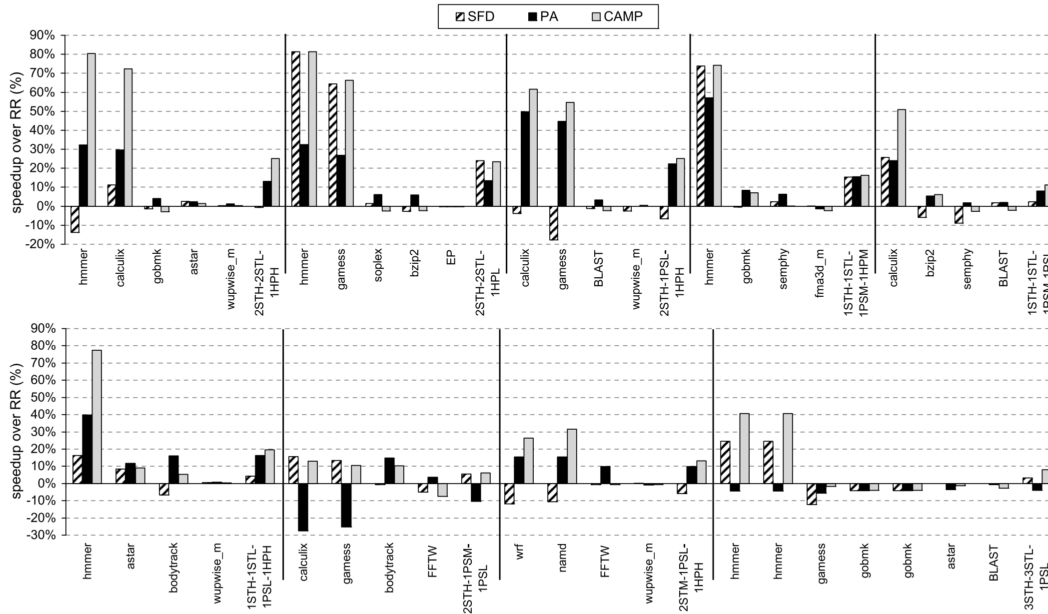


Fig. 16. Speedup of asymmetry-aware schedulers on 2FC-10SC.

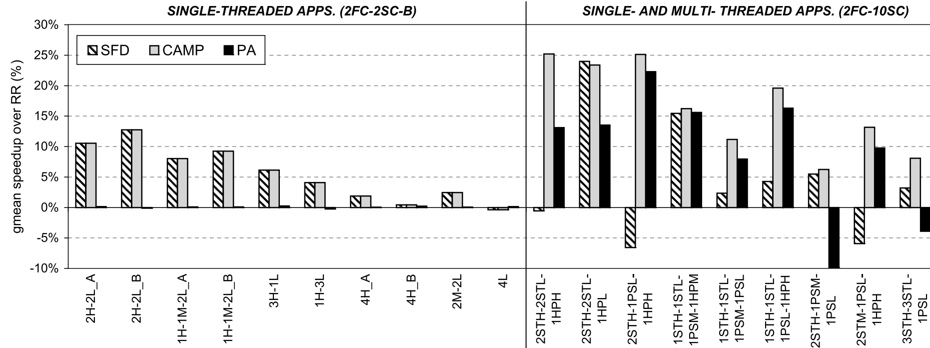


Fig. 17. Gmean speedup of SFD, PA and CAMP schedulers when running single-threaded and multi-threaded workloads on the Intel-12 platform.

configurations consisting of two chips (2FC-2SC-B and 2FC-10SC). As evident, CAMP reaps greater benefits than the other investigated schedulers across the board.

6. RELATED WORK

A large body of work has advocated the potential benefits of asymmetric single-ISA multicore processors over their symmetric counterparts [Annavaram et al. 2005; Hill and Marty 2008; Kumar et al. 2003, 2004; Mogul et al. 2008]. These benefits are concisely summarized in an article by Gillespie [2008] from Intel, where he lays out some of the background for why this shift towards asymmetric systems is likely to happen, describes the potential variations on the hardware architectures, and the distinct

challenges. OS scheduling is one of the most critical challenges, and this is the focus of this article.

Several asymmetry-aware schedulers were proposed in previous work. They delivered either efficiency or TLP specialization, but not both. In this section, we first introduce the most relevant scheduling proposals exploiting efficiency specialization; and then go on to describe the related work that demonstrated the effectiveness of TLP specialization.

Kumar et al. [2004] and Becchi and Crowley [2006] independently proposed similar schedulers that employed efficiency specialization, targeting workloads consisting of single-threaded applications only. For deciding which applications would use fast cores most efficiently, these schedulers relied solely on threads' speedup factors. A thread's speedup factor was *directly measured* online by running the thread on cores of both types and computing the observed performance ratio (IPS ratio). Algorithms proposed by Becchi and Kumar and were evaluated in a simulated environment. When real implementations of these algorithms were done as part of our earlier work [Shelepov et al. 2009], we found that the proposed methods for computing the speedup factor were inaccurate since applications may exhibit a nonuniform 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.

Both static and dynamic approaches have been explored in the research community to overcome the main problems associated to direct measurement of SFs. Along with the Heterogeneity-Aware Signature Supported (HASS) scheduler, we proposed a static approach that involved estimating SFs using offline-collected performance information about applications [Shelepov et al. 2009]. This approach relied on architectural signatures (i.e.: compact summaries of applications runtime properties). The information contained in an application's architectural signature (possibly embedded in its binary) enabled 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 overcame the difficulties associated with direct measurement of the SF, it had limitations of its own. For example, the method does not always allow to capture dynamic properties of the application since it relies on static information. At the same time, this method required co-operation from the developer to perform the steps needed for the generation of the architectural signature.

Koufaty et al. [2010], concurrently with and independently from us [2010a], devised a similar approach to estimate speedup factors online. Although this approach also relied on hardware performance counters to determine the relative speedup, the authors sought to find performance metrics that had a high correlation with the SF rather than modeling it accurately. A key contribution of the work by Koufaty et al. was the performance characterization of an emulated asymmetric system where cores differ in retirement width. They demonstrated that this form of asymmetry is more representative for performance asymmetry than than cores differing in processor frequency. In this paper, we explored several forms of performance asymmetry and demonstrated that, equipped with accurate platform-specific estimation models, the CAMP scheduler is able to reap significant benefits for a broad spectrum of multiapplication workloads.

We now switch the discussion to the related work that focused on exploiting AMPs to accelerate serial phases of parallel applications (aka TLP specialization). Hill and Marty [2008] and Morad et al. [2004] derived theoretical to predict the speedup for parallel applications with serial phases running on AMPs. The former group concluded that AMPs can potentially deliver significantly better performance than symmetric multicore processors for applications whose serial phases constituted at least 5% of

the execution time. The speedup models proposed in the aforementioned theoretical works cannot, however, replace the utility factor (UF) in our CAMP scheduler since the models do not account for the fact that applications may exhibit different speedup factors (SFs) over time, and were not designed to predict the speedup for different phases in the parallel application at runtime using exclusively runtime information accessible at the OS level. Annavaram et al. [2005] 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 (like the ones explored in this article).

In a previous work, we proposed the Parallelism-Aware (PA) scheduler, the first OS-level scheduler delivering TLP specialization [Saez et al. 2010b]. As shown in Section 5, PA is able to match the performance of CAMP in some multiapplication scenarios, but it fails to provide results comparable to CAMP's when the workload exhibits sufficient diversity of speedup factors. PA used the number of runnable threads as an approximation for the amount of parallelism in the application. 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. This approach is effective because applications typically let the unused threads block, perhaps after a short period of *spinning* (i.e., busy waiting). While blocking is coordinated with the OS, making it possible to detect phase transitions, spinning is not. Along with the PA scheduler, a *spin-then-notify* synchronization mode was also proposed in Saez et al. [2010b] to make spinning visible to the OS scheduler. Making the scheduler aware of which threads are actually spinning (thus not doing useful work) is also beneficial in the context of efficiency specialization, where high-SF threads are assigned preferentially to fast cores. Since spinning threads can achieve a very high speedup factor,[4] explicit notifications from the thread library or the runtime system on spinning threads would prevent the OS scheduler from mapping these threads to fast cores.

Other researchers explored additional techniques to accelerate sequential code on AMPs. For example, ACS is a system that was explicitly designed to accelerate lock-protected critical sections [Suleman et al. 2009]. ACS uses a combination of compiler and hardware techniques to ensure that the code inside the critical section is executed on a fast core. Thread migrations are performed entirely in hardware, so as opposed to our work, they did not address the design of an OS scheduler. Furthermore, like Annavaram's work, the system supports only single-application workloads. The authors indicate that operating system assistance would be required to support multiapplication workloads. Our work provides support similar to that which would be needed by ACS. Another work relevant to our research is on Feedback-Driven Threading (FDT) by Suleman et al. [2008]. In this work, the authors developed a runtime system that performs an online discovery of the optimal threading level for parallel applications. Of particular relevance to us is the discussion provided by the authors about the sources of scalability bottlenecks in parallel applications. Our CAMP scheduler addresses those bottlenecks that are due to load imbalance or data serialization. The authors also identify memory bandwidth as another possible bottleneck. With this type of bottleneck, there is not a well-defined serial phase, but the wait times due to memory bus contention are arbitrarily distributed among the threads. In this scenario, AMP systems coupled with CAMP policy are less appropriate to mitigate serial

[4]Spin loops use the CPU pipeline very efficiently. Best practices in implementing spinlocks dictate using algorithms where a thread spins on a local variable, which leads to a high instruction throughput.

bottlenecks, because there is not a well-defined serial phase that can be accelerated on a fast core. To address these types of bottlenecks, application-level solutions such as the aforementioned FDT are more suitable and effective.

While most scheduling proposals were concerned with TLP or efficiency specialization, a few researchers looked at less conventional types of specialization. Mogul et al. [2008] proposed using slow cores in asymmetric systems for executing system calls, since system calls are dominated by code that uses fast and powerful cores inefficiently. They modified an operating system to automatically migrate threads to low-power cores when they switch to kernel mode for the execution of certain system calls. The performance improvements achieved by Mogul's scheduler, however, were not very large due to the overhead associated with thread migrations aimed to execute system calls on slow cores. Our implementation of CAMP partially exploits this kind of core specialization since all Solaris's kernel threads are forced to run 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 [Balakrishnan et al. 2005], or keeping the load on fast cores higher than the load on slow cores [Li et al. 2007].

While existing 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. The CAMP scheduler makes it possible to close this gap.

7. CONCLUSIONS AND FUTURE WORK

In this article we have presented a comprehensive scheduling algorithm for asymmetric multicore processors. Although the advantages of exploiting the information on applications' ILP and TLP on AMPs were well understood before, no one had addressed the design of the corresponding *unified* scheduling 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 multithreaded applications, while PA is unable to compete with CAMP when applications exhibit a wide variety of speedup factors (i.e., relative speedup that a thread derives on a fast core relative to a slow one).

Key elements for the success of CAMP are the Utility Factor (UF) and its reliance on estimation models to approximate per-thread speedup factors (SFs). In this work, we devised SF estimation models for different forms of performance asymmetry. These models make it possible for CAMP to reap benefits in diverse asymmetric multicore configurations.

An interesting direction for future work is to study the interaction between migrations and caching behavior. Numerous studies have shown that some applications are more sensitive to cross-memory-domain migrations than others due to the nature of their memory-access patterns [Constantinou et al. 2005; Li et al. 2007], and if threads share cached data the problem becomes even more challenging [Tam et al. 2007]. Incorporating cache awareness into asymmetry-aware algorithms like CAMP would be a first step toward designing an all-encompassing scheduling algorithm for asymmetric multicore systems.

APPENDIX

In this section we provide a detailed derivation process for the $Analytical_{UF}$ formula, which approximates the speedup that a parallel application obtains from the BusyFCs scheduler, relative to an execution where only slow cores are used.

In constructing the model, we make several simplifying assumptions about the nature of the parallel application. First, we assume that the application is *perfectly balanced*, namely, all its threads perform the same amount of work in parallel until completion. Second, the application consists of k balanced parallel phases, $k \geqslant 1$, separated by global synchronization points (e.g., barriers). In a parallel phase P, all threads complete the same number of instructions (NI_P) in parallel (the number of completed instructions may differ across parallel phases, though). The latter assumption indirectly states that all threads will execute the same number of instructions till completion. More precisely, if NI denotes the total number of instructions executed by any thread then $NI = \sum_{i=1}^{k} NI_i$.

We begin by introducing some auxiliary notation:

— $CT_{BusyFCs}$: Completion time of the application under BusyFCs.
— CT_{slow} : Completion time of the application when using slow cores only.
— N_{FC}, N_{SC} : Number of fast and slow cores of the AMP system, respectively.
— $N_{threads}$: Total number of threads the parallel application runs with.
— $NI_{fc,i}, NI_{sc,i}$: The total number of instructions that the *slowest* thread completes on fast and slow cores in the i-th parallel phase under BusyFCs, respectively.
— SPI_{fc}, SPI_{sc} : Average number of seconds per instruction[5] on fast and slow cores.
— SF_{avg} : Average speedup factor (SF) of all threads in the application, where $SF_{avg} = \frac{SPI_{sc}}{SPI_{fc}}$.

As stated in Section 3.1, we can approximate CT_{slow} with the time that any thread in the application (with $SF = SF_{avg}$) takes to execute all its instructions on a slow core:

$$CT_{slow} = NI \cdot SPI_{sc} = \sum_{i=1}^{k}(NI_i \cdot SPI_{sc}). \tag{3}$$

Determining the completion time of the application under BusyFCs ($CT_{BusyFCs}$) entails approximating the number of instructions that the *slowest thread* executes on fast and slow cores for each parallel phase. The formula for $CT_{BusyFCs}$ is as follows:

$$CT_{BusyFCs} = \sum_{i=1}^{k}(NI_{fc,i} \cdot SPI_{fc} + NI_{sc,i} \cdot SPI_{sc}). \tag{4}$$

Determining $NI_{fc,i}$ and $NI_{sc,i}$ requires analyzing the behavior of the BusyFCs scheduler during the execution of parallel phase of the application. Recall that the sequence of actions performed by BusyFCs during the execution of a parallel phase is outlined in Algorithm 1.

To facilitate the analysis we introduce additional notation. Let $F_{fc}(j)$ be the fraction of instructions that threads running on fast cores in the j-th iteration of Algorithm 1 complete during that iteration (on fast cores). Conversely, let $F_{sc}(j)$ be the fraction of

[5]We opted to use the SPI instead of the number of cycles per instruction (CPI) or instructions per second (IPS) since this makes it possible to simplify calculations considerably in the derivation process.

Table VIII. Fraction of Instructions Executed by Threads Scheduled on Fast and Slow Cores in Different Iterations of Algorithm 1

Iteration	Fast cores $\longrightarrow F_{fc}(j)$	Slow cores $\longrightarrow F_{sc}(j)$
#1	1	$\frac{1}{SF_{avg}}$
#2	$1-\frac{1}{SF_{avg}}$	$\frac{1}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}$
#3	$1-\left(\frac{1}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}\right)$	$\frac{1}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}}{SF_{avg}}$
...	...	...
# j	$1-F_{sc}(j-1)$	$F_{sc}(j-1)+\frac{F_{fc}(j)}{SF_{avg}}$

instructions executed on slow cores from the beginning of the parallel phase by those threads that still remain assigned to slow cores in the j-th iteration of the algorithm.

Table VIII displays the values of $F_{fc}(j)$ and $F_{sc}(j)$ for each iteration of Algorithm 1. Suppose that each thread has to complete NI_P instructions to reach the synchronization barrier. In the first iteration of Algorithm 1, threads running on fast cores complete all their instructions NI_P, while threads on slow cores can complete $\frac{NI_P}{SF_{avg}}$ instructions. Equivalently, these values can be expressed in terms of the fraction over NI_P, as 1 (100%) and $1/SF_{avg}$, respectively. Likewise, threads assigned to fast cores in the second iteration have to complete their remaining fraction of instructions: $1-\left(1/SF_{avg}\right)$. This works out to that value because those threads already completed a fraction of $1/SF_{avg}$ on slow cores in the previous iteration. In a general case, $F_{fc}(j)$ and $F_{sc}(j)$ can be defined mutually recursively as follows:

$$F_{fc}(j)=\begin{cases}1 & j=1\\ 1-F_{sc}(j-1) & j>1\end{cases}\qquad F_{sc}(j)=\begin{cases}\frac{1}{SF_{avg}} & j=1\\ F_{sc}(j-1)+\frac{F_{fc}(j)}{SF_{avg}} & j>1\end{cases}. \tag{5}$$

We go on to obtain a nonmutually recursive formula for $F_{fc}(j)$ as follows:

$$\left.\begin{aligned}F_{sc}(j-1) &= 1-F_{fc}(j)\\ F_{sc}(j) &= 1-F_{fc}(j+1)\\ F_{sc}(j) &= F_{sc}(j-1)+\frac{F_{fc}(j)}{SF_{avg}}\end{aligned}\right\}\Longrightarrow F_{fc}(j)=F_{fc}(j-1)\cdot\left(1-\frac{1}{SF_{avg}}\right). \tag{6}$$

We proceed to unroll Equation (6) and obtain a nonrecursive formula:

$$\begin{aligned}F_{fc}(j) &= F_{fc}(j-1)\cdot\left(1-\frac{1}{SF_{avg}}\right)=F_{fc}(j-2)\cdot\left(1-\frac{1}{SF_{avg}}\right)\cdot\left(1-\frac{1}{SF_{avg}}\right)\\ &= F_{fc}(1)\cdot\underbrace{\left(1-\frac{1}{SF_{avg}}\right)\cdot\left(1-\frac{1}{SF_{avg}}\right)\cdot\ldots\cdot\left(1-\frac{1}{SF_{avg}}\right)}_{j-1}\\ &= 1\cdot\underbrace{\left(1-\frac{1}{SF_{avg}}\right)\cdot\left(1-\frac{1}{SF_{avg}}\right)\cdot\ldots\cdot\left(1-\frac{1}{SF_{avg}}\right)}_{j-1}=\left(1-\frac{1}{SF_{avg}}\right)^{j-1}.\end{aligned} \tag{7}$$

At this point, we go on to approximate the time that the *slowest* thread takes to execute all its instructions in a parallel phase, which, as stated in Section 3.1, determines the critical path of the parallel phase and, in turn, the performance of the application. Under BusyFCs, the slowest thread in a given parallel phase is the one mapped to a fast core in the last iteration of Algorithm 1. (There may be several threads running on fast cores in the last iteration, but any of them will complete roughly at the same time.) Therefore, determining the completion time of a given parallel phase boils down to computing the number of instructions executed on fast and slow cores by threads mapped to fast cores in the last iteration. More precisely, let $NI_{fc,i}$ and $NI_{sc,i}$ be the number instructions executed by the slowest thread in the i-th parallel phase on fast and slow cores respectively, so we have that:

$$NI_{fc,i} = NI_i \cdot F_{fc}(N_{iter}). \tag{8}$$

$$NI_{sc,i} = NI_i \cdot (1 - F_{fc}(N_{iter})). \tag{9}$$

$$\text{where} \quad N_{iter} = \left\lfloor \frac{N_{threads} - 1}{N_{FC}} \right\rfloor + 1.$$

Finally, we approximate the theoretical speedup of the application under BusyFCs with respect to an execution where only slow cores are used, as follows:

$$Analytical_{UF} = \frac{CT_{slow}}{CT_{BusyFCs}} = \frac{\sum_{i=1}^{k}(NI_i \cdot SPI_{sc})}{\sum_{i=1}^{k}(NI_{fc,i} \cdot SPI_{fc} + NI_{sc,i} \cdot SPI_{sc})} =$$

$$= \frac{\sum_{i=1}^{k}(NI_i \cdot SPI_{fc} \cdot SF_{avg})}{\sum_{i=1}^{k}(NI_{fc,i} \cdot SPI_{fc} + NI_{sc,i} \cdot SPI_{fc} \cdot SF_{avg})} = \frac{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i \cdot SF_{avg})}{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_{fc,i} + NI_{sc,i} \cdot SF_{avg})}$$

$$= \frac{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i \cdot SF_{avg})}{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i \cdot F_{fc}(N_{iter}) + NI_i \cdot (1 - F_{fc}(N_{iter})) \cdot SF_{avg})}$$

$$= \frac{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i) \cdot SF_{avg}}{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i) \cdot (F_{fc}(N_{iter}) + (1 - F_{fc}(N_{iter})) \cdot SF_{avg})}$$

$$= \frac{SF_{avg}}{F_{fc}(N_{iter}) + (1 - F_{fc}(N_{iter})) \cdot SF_{avg}} = \frac{SF_{avg}}{F_{fc}(N_{iter}) \cdot (1 - SF_{avg}) + SF_{avg}}$$

$$= \frac{SF_{avg}}{\left(1 - \frac{1}{SF_{avg}}\right)^{N_{iter}-1} \cdot \left(1 - SF_{avg}\right) + SF_{avg}} = \frac{SF_{avg}}{\left(1 - \frac{1}{SF_{avg}}\right)^{\left\lfloor \frac{N_{threads}-1}{N_{FC}} \right\rfloor} \cdot \left(1 - SF_{avg}\right) + SF_{avg}}.$$

Hence, the resulting formula for the theoretical speedup under BusyFCs is as follows:

$$Analytical_{UF} = \frac{SF_{avg}}{\left(1 - \frac{1}{SF_{avg}}\right)^{\left\lfloor \frac{N_{threads}-1}{N_{FC}} \right\rfloor} \cdot \left(1 - SF_{avg}\right) + SF_{avg}}. \qquad (10)$$

REFERENCES

ANNAVARAM, M., GROCHOWSKI, E., AND SHEN, J. 2005. Mitigating Amdahl's law through EPI throttling. In *Proceedings of the International Symposium on Computer Architecture (ISCA'05)*. 298–309.

ARM. 2011. Big.LITTLE Processing with ARM CortexTM-A15 & Cortex-A7. White paper, http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf.

BALAKRISHNAN, S., RAJWAR, R., UPTON, M., AND LAI, K. 2005. The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Comput. Architect. News 33*, 2, 506–517.

BECCHI, M. AND CROWLEY, P. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the International Conference on Computing Frontiers (CF'06)*. 29–40.

BLAGODUROV, S., ZHURAVLEV, S., AND FEDOROVA, A. 2010. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst. 28*, 8:1–8:45.

CONSTANTINOU, T., SAZEIDES, Y., MICHAUD, P., FETIS, D., AND SEZNEC, A. 2005. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Architect. News 33*, 80–91.

FREEH, V. W., LOWENTHAL, D. K., PAN, F., KAPPIAH, N., SPRINGER, R., AND ROUNTREE, B. L. 2007. Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Trans. Parall. Distrib. Syst. 18*, 6, 835–848.

FRIEDMAN, J. H. 1999. Stochastic gradient boosting. www-stat.stanford.edu~jhf/ftp/stobst/pdf.

GILLESPIE, M. 2008. Preparing for the second stage of multi-core hardware: Asymmetric (heterogeneous) cores. Intel white paper.

HALL, M., FRANK, E., HOLMES, G., PFAHRINGER, B., REUTEMANN, P., AND WITTEN, I. H. 2009. The WEKA data mining software: An update. *SIGKDD Explor. Newsl. 11*, 10–18.

HILL, M. D. AND MARTY, M. R. 2008. Amdahl's law in the multicore era. *IEEE Comput. 41*, 7, 33–38.

KOUFATY, D., REDDY, D., AND HAHN, S. 2010. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of Eurosys'10*.

KUMAR, R., FARKAS, K. I., JOUPPI, N., ET AL. 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'03)*.

KUMAR, R., TULLSEN, D. M., RANGANATHAN, P., ET AL. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the International Symposium on Computer Architecture (ISCA'04)*.

LI, T., BAUMBERGER, D., KOUFATY, D., ET AL. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the Conference on Supercomputing (SC'07)*. 1–11.

LI, T., BRETT, P., KNAUERHASE, R., KOUFATY, D., REDDY, D., AND HAHN, S. 2010. Operating system support for overlapping-ISA heterogeneous multicore architectures. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. 1–12.

MOGUL, J. C., MUDIGONDA, J., BINKERT, N., RANGANATHAN, P., AND TALWAR, V. 2008. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro 28*, 3, 26–41.

MORAD, T., WEISER, U., AND KOLODY, A. 2004. ACCMP—Asymmetric cluster chip multi-processing. CCIT Tech. rep #448.

SAEZ, J. C., FEDOROVA, A., PRIETO, M., ET AL. 2010a. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of Eurosys'10*. 139–152.

SAEZ, J. C., FEDOROVA, A., PRIETO, M., ET AL. 2010b. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. In *Proceedings of the International Conference on Computing Frontiers (CF'10)*. 31–40.

SAEZ, J. C., SHELEPOV, D., FEDOROVA, A., AND PRIETO, M. 2011. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. *J. Parall. Distrib. Comput. 71*, 114–131.

SHELEPOV, D., SAEZ, J. C., JEFFERY, S., ET AL. 2009. HASS: A scheduler for heterogeneous multicore systems. *ACM SIGOPS Op. Syst. Rev. 43*, 2, 66–75.

SULEMAN, M. A., MUTLU, O., QURESHI, M. K., AND PATT, Y. N. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. 253–264.

SULEMAN, M. A., QURESHI, M. K., AND PATT, Y. N. 2008. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. *SIGARCH Comput. Architect. News 36*, 1, 277–286.

TAM, D., AZIMI, R., AND STUMM, M. 2007. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of EuroSys'07.* 47–58.

VAN DER PAS, R. 2005. The OMPlab on Sun Systems. In *Proceedings of the International Workshop on OpenMP (IWOMP'05)*.

Received March 2011; revised December 2011; accepted January 2012