Contention-Aware Fair Scheduling for Asymmetric Single-ISA Multicore Systems

Adrian Garcia-Garcia, Juan Carlos Saez, and Manuel Prieto-Matias

Abstract—Asymmetric single-ISA multicore processors (AMPs), which integrate high-performance big cores and low-power small cores, were shown to deliver higher performance per watt than symmetric multicores. Previous work has demonstrated that the OS scheduler plays an important role in realizing the potential of AMP systems. While throughput optimization on AMPs has been extensively studied, delivering fairness on these platforms still constitutes an important challenge to the OS. To this end, the scheduler must be equipped with a mechanism enabling to accurately track the progress that each application in the workload makes as it runs on the various core types throughout the execution. In turn, this progress largely depends on the benefit (or speedup) that an application derives on a big core relative to a small one, which may differ greatly across applications. While existing fairness-aware schedulers take application relative speedup into consideration when tracking progress, they do not cater to the performance degradation that may occur naturally due to contention on shared resources among cores, such as the last-level cache or the memory bus. In this paper, we propose CAMPS, a contention-aware fair scheduler for AMPs that primarily targets long-running compute-intensive workloads. Unlike other schemes, CAMPS does not require special hardware extensions or platform-specific speedup-prediction models to function. Our experimental evaluation, which leverages real asymmetric hardware and scheduler implementations in the Linux kernel, demonstrates that CAMPS improves fairness by up to 11 percent with respect to a state-of-the-art fairness-aware OS-level scheme, while delivering better system throughput.

Index Terms—Asymmetric multicore, scheduling, operating systems, fairness, shared resource contention, Linux kernel

1 Introduction

Over the last years, two major trends have arisen in microprocessor design and manufacturing: the integration of an increasing number of cores per chip, and coupling different core types on the same platform for diverse and specialized use. The second trend has given rise to growing interest in heterogeneous architectures and system configurations.

The degree of diversity segregates heterogeneous architectures into various classes, each becoming a unique point in the design space [1], [2]. One extreme of this spectrum is to couple a modest number of high-performance cores with accelerators [3] or with special-purpose processing units. This is the case of systems such as the IBM Cell Broadband Engine [4] or CPU-GPU platforms, where the various cores typically expose a different Instruction Set Architecture (ISA). Despite their benefits, these architectures usually require substantial programming effort [2], [5]. This kind of heterogeneous platforms stands in contrast with asymmetric single-ISA multicore processors (AMPs) [6], which integrate a mix of complex high-performance big cores and power-efficient small cores. Our work targets this type of heterogeneous architecture, whose memory hierarchy organization may come in a variety of forms. While an application can be designed so as to explicitly exploit the features of the different core types of an AMP in a unified and dedicated fashion, the shared ISA and general-purpose nature of these cores allows the execution of asymmetry-agnostic (unmodified) software. This versatility, coupled with the outstanding energy efficiency benefits of AMP designs [6], has drawn the attention of major hardware players, giving rise to the ARM big.LITTLE processor [7], [8], or to the Intel QuickIA [9].

While applications specifically designed to leverage the capabilities of the various cores in an AMP can be effectively run by binding the various threads/tasks to the core type where they are meant to run (e.g., via affinity masks), delivering the potential of AMPs to unmodified applications (the ones we target) poses a number of challenges to the operating system [1], [2], some of which must be properly addressed by the scheduler [10], [11]. One important challenge is how to effectively distribute big-core cycles among the various applications in a workload. Most scheduling schemes proposed for AMPs have been designed to optimize the system throughput for multiprogram workloads [10], [12], [13], [14]. To make this happen, the scheduler must devote big cores to running those applications that use big cores efficiently, since they derive performance improvements (speedup) relative to running on small cores [6]. Additional throughput gains can be obtained by using big cores to accelerate scalability bottlenecks present in multithreaded programs [13], [15], [16], [17], [18].

Unfortunately, asymmetry-aware schedulers that strive to optimize throughput alone are known to be inherently unfair [11]. Unfairness gives rise to a number of undesirable

• *The authors are with the Faculty of Computer Science and Engineering, Complutense University of Madrid, Madrid 28040, Spain. E-mail: {adriagar, jcsaezal, mpmatias}@ucm.es.*

Manuscript received 12 July 2017; revised 21 Apr. 2018; accepted 7 May 2018. Date of publication 17 May 2018; date of current version 7 Nov. 2018. (Corresponding author: Juan Carlos Saez.) Recommended for acceptance by G. Heiser. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2018.2836418

effects on the system [19], [20]. For example, when using a scheduler that attempts to optimize throughput only in the AMP, an application's completion time may differ significantly across executions, depending on its co-runners in the workload [11]. Moreover, equal-priority applications may not experience the same performance degradation when running together relative to the performance observed when each application runs alone on the AMP. These issues make priority-based scheduling policies ineffective [20], reduce performance predictability [2], [21], [22] and may lead to wrong billings in commercial cloud-like computing services, where users are charged for CPU hours [19].

Our work primarily explores how to fairly schedule, at the OS level, a mix of unmodified applications on an AMP system. To this end, the scheduler must even out the progress made by the various applications as they run on the different core types throughout the execution [11], [23]. To do so, the scheduler must be equipped with a mechanism enabling to measure the performance degradation accumulated by an application at runtime with respect to its solo execution (aka. *slowdown*). On AMPs, the slowdown depends on two main factors: (1) *performance asymmetry* and (2) *shared-resource contention*. *Performance asymmetry* refers to the fact that most applications derive a non-negligible speedup from using high-performance big cores relative to running on low-power small ones. When a thread runs on a small core, it slows down in proportion to its big-to-small speedup, which may differ greatly across applications and may vary over time through different program phases [24]. *Shared-resource contention* may also lead to substantial performance degradation. In current AMP hardware, clusters of cores of the same type (big or small) typically share a last-level cache [8], [9], [25] and other memory-related resources with the remaining cores, such as the DRAM controller. Applications running on the various cores may compete with each other for these shared resources, which could degrade their performance in an uneven and unpredictable way, as the hardware itself does not guarantee a fair usage of these resources [20], [21], [26], [27], [28], [29].

Recent scheduling proposals for AMPs, such as Equal-Progress [23] or ACFS [11], attempt to enforce fairness by just catering to performance asymmetry aspects, but they do not take shared-resource contention effects into account. As we demonstrate in this work, this leads to substantial performance/fairness degradation when several memory-intensive applications are present in the workload. Conversely, contention-conscious approaches that aim to deliver fairness [22], [27], [28] or strive to improve performance isolation [21], [30], [31] are not designed to work on systems that combine high-performance cores with low-power cores with different microarchitectural features. Hence, these schemes do not factor in performance asymmetry.

To fill this gap, we propose CAMPS, an OS-level contention-aware scheduler for AMPs, which seeks to optimize fairness while maintaining acceptable system throughput. Our scheduler also exposes a configurable parameter enabling the user to trade fairness for throughput. As the vast majority of schedulers proposed for AMPs [10], [11], [13], [14], [16], [23], [24], our proposal primarily targets long-running compute-intensive workloads. Notably, in this paper we build on our prior work [11] to advance the state of the art in fairness-oriented asymmetry-aware scheduling, by now factoring in shared-resource contention effects (our previous approach [11] is contention unaware). In particular, our paper makes the following main contributions:

- We devised a novel runtime mechanism to predict the slowdown that a thread in the workload experiences as it runs on the various cores of an AMP. Specifically, our scheduler approximates the current slowdown by monitoring various runtime metrics via performance monitoring counters (PMCs), and by comparing that information with the thread's past history gathered in low contention scenarios.
- Unlike other OS-level schedulers for AMPs [10], [11], CAMPS does not rely on platform-specific speedup prediction models, which typically entail the monitoring of a specific set of hardware PMC events that may differ substantially across processor models and architectures [10], [11], [13], [32]. Instead, our proposal employs a small and fixed set of performance metrics that can be easily gathered using PMCs available in commercial AMP hardware, thus making the scheduler portable across architectures.
- We implemented CAMPS in the Linux kernel, on top of the Completely Fair Scheduler (CFS), which is largely asymmetry agnostic. As we demonstrate in this work, the completion time of an application under the stock Linux scheduler may vary substantially across multiple runs of the same workload on an AMP. As a result, CFS and the HMP (Heterogeneous Multi-Processing) scheduler [33] –an extension of CFS for big.LITTLE platforms– constitute unfair scheduling schemes for asymmetric multicores, especially when compute-intensive applications are present in the workload. Notably, CAMPS delivers more consistent performance from run to run and higher degree of fairness for a wider spectrum of workloads.
- For our experimental evaluation, we employed the Intel QuickIA prototype [9] as well as commercial ARM-based asymmetric multicore platforms [8], [25]. We performed an extensive experimental comparison with previously proposed asymmetry-aware schemes [10], [11], [13]. Our analysis reveals that CAMPS improves fairness by up to 11 percent compared to a state-of-the-art fairness-aware scheduler [11], and at the same time improves throughput by up to 17 percent.

The rest of the paper is organized as follows. Section 2 motivates our work. Section 3 discusses related work. Section 4 outlines the design of CAMPS. Section 5 showcases our experimental results and Section 6 concludes.

2 Motivation

In this section we first introduce the notion of fairness employed in our work, and discuss the challenges associated with determining the slowdown at runtime. We then present an experimental study that showcases the main observation we exploit to determine the slowdown on-line on AMPs.

2.1 Fairness on AMPs

Previous research on fairness for CMPs [19], [20] and AMPs [16], [23] define a scheme as fair if equal-priority applications in a workload suffer the same slowdown due to sharing the system. To cope with this notion of fairness, we turned to the *unfairness* metric, which has been extensively used in previous work [11], [16], [19], [20], [27]. This lower-is-better metric is defined as follows:

$$Unfairness = \frac{MAX(Slowdown_1, ..., Slowdown_n)}{MIN(Slowdown_1, ..., Slowdown_n)}, \qquad (1)$$

where n is the number of applications in the workload and $Slowdown_i = \frac{CT_{sched,i}}{CT_{alone,i}}$. In turn, $CT_{sched,i}$ denotes the completion time of application i under a given scheduler, and $CT_{alone,i}$ is the completion time of application i when running alone on the AMP (with all the big cores available).

The slowdown of an individual thread (or that of a single-threaded application) observed during a certain execution phase can be defined in terms of the number of instructions per second (*IPS*) as follows:

$$Slowdown = IPS_{alone}/IPS_{sched}, \qquad (2)$$

where IPS_{alone} is the number of instructions per second observed for the specific phase when the thread runs alone on the system, and IPS_{sched} denotes the IPS achieved by the thread when it runs the same execution phase, but in a program mix under a given scheduling algorithm.

In this work, we assume that the IPS_{alone} on an AMP is maximized when a thread runs on a big core in isolation; that is the case across all the applications explored in our experimental platforms. We should also highlight that in multithreaded programs, the IPS can be a somewhat misleading performance metric, since a thread can exhibit a high IPC when busy waiting (*spinning*) in synchronization primitives (e.g., barriers). To make the OS scheduler aware of these situations, where threads do no useful work, our proposal leverages *spin notifications* from the user-level runtime system. We elaborate on this aspect in Section 4.5.

Delivering fairness entails ensuring that the slowdown accumulated by the various application threads throughout the execution remains as even as possible [11], [20], [23], [27], while maintaining acceptable throughput. To this end, the scheduler must be equipped with a mechanism to determine a thread's slowdown online. Notably, measuring the slowdown at runtime by using Eq. (2) is difficult in practice; while a thread's IPS_{sched} can be easily obtained via PMCs, accurately determining IPS_{alone} online is a challenging task, even on symmetric CMPs [27], [28]. For that reason, existing scheduling algorithms for symmetric CMPs typically rely on estimation models to approximate IPS_{alone} [27], or employ heuristics to determine the degree of performance degradation indirectly via contention-related metrics [28], such as the last-level cache (LLC) miss rate [26], [27]. Unfortunately, these scheduling algorithms are not designed to work on systems featuring multiple core types. Moreover, adapting them to AMP systems is difficult, as these schedulers assume that the value of key performance metrics used to drive scheduling decisions (e.g., IPC or LLC miss rate) do not vary across cores when the application runs alone. On current AMP hardware [7], [8], [9], this assumption is not valid, as cores may exhibit different microarchitectural features and cache sizes [10], [11]. This fact further complicates determining the slowdown on an AMP.

2.2 Impact of Shared Resource Contention on AMPs

Recently proposed fairness-aware schedulers for AMPs [11], [23] implicitly rely on the assumption that a thread's slowdown is 1 (no performance degradation) when it runs on a big core, even if it runs simultaneously with other threads. In a similar vein, the thread's big-to-small performance ratio –also referred to as the *speedup factor* (SF) [13]– is used by these schedulers to approximate the slowdown when the thread runs on a small core. (The SF can be determined online by various means, as we will discuss in Section 3.)

Assuming that a thread's slowdown is negligible when it runs on a big core (as done in [11], [23]) is unrealistic under shared resource contention. To illustrate this fact we experimented with diverse AMP platforms. Our analysis allowed us to draw two major insights:

(1) Performance degradation due to resource sharing among big cores can be substantial on current AMP hardware (up to 2.98x on our experimental platforms), and should be accounted for when tracking the slowdown of a thread to enforce fairness as well as to ensure effective utilization of big cores. In fact, for some programs, the benefit that comes from running on a big core w.r.t. a small one could be substantially reduced due to the contention-related performance degradation.

(2) Monitoring a thread's IPS when it runs on a big core, and in a low contention scenario across the neighboring big cores (i.e., those sharing the LLC) is typically a good estimate for IPS_{alone}. Essentially, the performance penalty that a thread mapped to a big core may suffer from placing multiple memory-intensive aggressors on small cores is usually very low compared to the one that comes from interference with memory-intensive threads running on neighboring big cores. This has to do with the memory-hierarchy organization on current AMP hardware, as well as with the fact that small cores typically utilize less memory bandwidth than big cores.

We draw these conclusions from experiments in which we measured the slowdown experienced by SPEC CPU applications when mapped to a big core, and running simultaneously with several instances of a memory-intensive aggressor application. We used benchmarks from both the CPU2000 and CPU2006 suites to consider a wider diversity of working set sizes. As the aggressor, we used the `bandwidth` benchmark [29], which introduces substantial contention on the LLC, shared buses and DRAM controller. On our platforms, we observed that `bandwidth` causes even a higher degree of contention than that generated by highly memory-intensive programs from SPEC CPU, such as `lbm`.

For our experiments, we used two ARM platforms: the ARM Juno [8] and the Odroid XU4 [25] boards, which are equipped with a 64-bit and a 32-bit big.LITTLE processor respectively. We also experimented with the Intel QuickIA prototype [9]. On these AMP platforms, which integrate a mix of high-performance out-of-order cores and low-power in-order cores, SPEC CPU benchmarks exhibit a wide range of big-to-small speedups: 1.55x-4.44x (Juno), 1.36x-6.63x (Odroid) and 1.02x-4.7x (QuickIA). Fig. 1 depicts the memory hierarchy organization as well the number of cores of

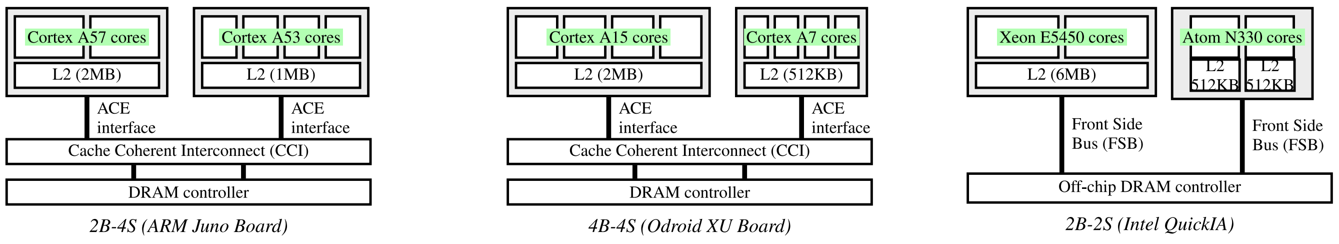

Fig. 1. Asymmetric multicore configurations.

each kind in the AMP configurations explored. For simplicity, we employ the nB-mS notation to refer to each configuration, where n and m denote the number of big and small cores, respectively. On 2B-4S and 4B-4S, the set of cores of the same type (big or small), which make up a cluster, share a last-level cache (L2) and a bus interface (AMBA) with the remaining cores in the cluster. On 2B-2S (a dual-socket system) a bus interface (FSB) is shared between cores of the same cluster; a shared LLC exists on the big core cluster only. All platforms feature a single DRAM controller.

Fig. 2 shows the slowdown (relative to the solo execution) that different applications experience when running simultaneously with several instances of `bandwidth`. For each benchmark, which is always assigned to a big core in our experiments, we explored different scenarios. In the first one, denoted as “1-aggressor-big”, the benchmark runs simultaneously with one instance of `bandwidth`, which is mapped to a different big core; the small cores remain idle in this case. In the other scenarios, labeled as “N-aggressors-small”, N instances of `bandwidth` are mapped to small cores; thus, in leaving the remaining big cores unused, we remove contention on the LLC and the bus interface of the big core cluster, but not on the DRAM controller.

As is evident, the slowdown can be substantial (up to 1.9x on 2B-4S, up to 2.65x on 4B-4S, and up to 2.98x on 2B-2S) when both the benchmark and a single aggressor instance run simultaneously on big cores, even though small cores are unused. By contrast, when one aggressor runs on a small core the slowdown drops significantly; for most benchmarks it is below 10 percent in this case. Actually, if we populate all the

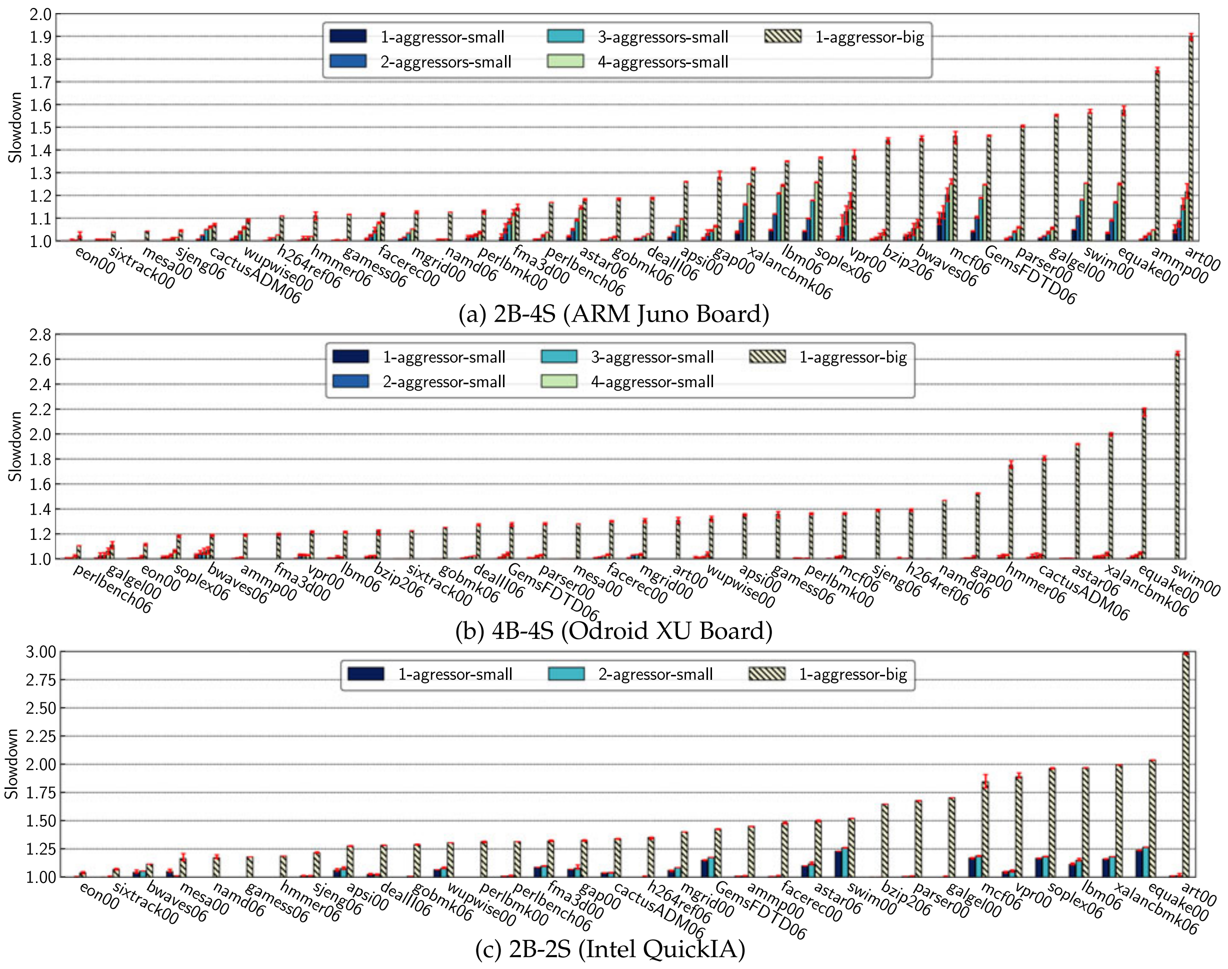

Fig. 2. Average slowdown relative to solo execution experienced by various benchmarks when mapped to a big core and run simultaneously with several instances of an aggressor. The error bars report the minimum and maximum values gathered across the various runs (five executions). A suffix (“00” or “06”) was appended to the application name to indicate the benchmark suite it belongs to (CPU2000 or CPU2006, respectively).

small cores with aggressor instances, slowdown values are no greater than 26 percent, still much smaller than those obtained when the aggressor runs on a big core. There are two main reasons associated with this behavior. First, the contention on the LLC and on the shared bus (big-core cluster) is removed completely in the "N-aggressors-small" scenarios. Second, we observed that the pressure a single aggressor puts on the shared resources is higher when it runs on a big core than on a small one. This has to do with the fact that in-order small cores cannot handle multiple outstanding cache misses, leading to a lower bus and memory bandwidth utilization, and as a result to a smaller degree of contention.

Finally, we should highlight that not every application experiences noticeable slowdown due to contention when executed together with highly memory-intensive benchmarks such as `bandwidth`. For example, this is the case of `sixtrack`, `eon` or `mesa`. As pointed out in previous work [26], [28], CPU-intensive applications with a very small working set (which fits in a reduced portion of the LLC) and good cache locality, or those that do not use the memory hierarchy substantially, do not experience significant performance penalty due to contention. As in [27], our scheduling proposal uses the bus transfer rate (BTR) to identify scenarios where threads are unlikely to suffer from contention when running on a big core cluster. In our platforms, the BTR is measured as follows:

$$\frac{bus\_read\_accesses * LLC\_cache\_line\_size * processor\_frequency}{total\_cycle\_count}.$$

3 Related Work

A large body of work has advocated the benefits of asymmetric multicores over symmetric CMPs [6], [34]. Despite these benefits, AMP systems pose significant challenges to the OS scheduler [1], [2]. Delivering fairness constitutes an important challenge, and this is the focus of our article.

As stated earlier, our OS-level scheduling proposal primarily targets long-running compute-intensive workloads, just like many other earlier schemes [10], [11], [13], [14], [23], [24]. Other authors have devised specialized schedulers to properly deal with other kinds of workloads on AMPs, such as latency-sensitive applications [35], [36], programs with irregular non-scalable parallelism [18] or multimedia applications [37]. As opposed to our proposal, none of these approaches strives to optimize fairness on AMPs by taking shared-resource contention effects into consideration.

Recent research has highlighted that fairness, system throughput and energy efficiency are largely conflicting optimization goals on AMPs [38]. In particular, optimizing fairness usually comes at the expense of degrading throughput and energy efficiency substantially [38]. Notably, to optimize any of the aforementioned aspects, the scheduler must consider the *speedup factor* of the various threads when making decisions [10], [11], [13], [14], [38]. The SF is defined as $\frac{IPS_{big}}{IPS_{small}}$, where IPS_{big} and IPS_{small} are the thread's instructions per second (IPS) ratios achieved on big and small cores respectively when running alone on the system.

Determining the SF. Three techniques have been explored to obtain the speedup factor online. The first approach is to measure the SF directly [6], [24], which requires running each thread on big and small cores to track the IPC on both core types. Previous work has shown that this approach, also known as *IPC sampling*, is subject to inaccuracies that naturally come from using IPC values from different program phases to approximate the SF [11], [12]. The second approach relies on *predicting a thread's SF* using its runtime properties collected on the *current* core type using PMCs [10], [11], [13], [32]. The main limitation of this approach is that it requires building an estimation model specifically tailored to the AMP platform in question. This entails conducting an offline analysis on each platform making it possible (1) to identify the set of performance events and metrics that turn suitable for SF prediction [10], [13], and (2) to determine the value of the various model coefficients (e.g., when using regression-based approaches [11], [13], [32]). The third technique is Performance Impact Estimation (PIE) [14], a hardware-aided mechanism that has been shown to provide accurate SF estimates. Unfortunately, PIE has not yet been adopted in commercial AMP platforms.

Our scheduling proposal predicts a thread's cross-core relative performance by measuring its actual IPS, and by comparing it with an estimate of the IPS_{alone} –approximated with big-core IPS values collected for different program phases in low-contention scenarios. This makes it possible to avoid the phase-related inaccuracies of IPC sampling [12], and allows us to cater to the potentially high variability of the IPS under different contention levels. Notably, CAMPS does not employ platform-specific SF prediction models, but instead relies on the monitoring of a fixed set of high-level performance metrics (the same across platforms), as explained in Section 4. This removes the need for conducting non-trivial offline analyses on each system to build speedup prediction models, thus improving the scheduler portability.

Fairness on AMPs. The first fairness-aware scheduler for AMPs was an asymmetry-aware Round-Robin (RR) scheme that simply fair-shares big cores among applications by triggering periodic thread migrations [24]. Fair-sharing big cores has proven to provide better performance and more repeatable completion times across runs on AMPs [2], [15] than default schedulers in general-purpose OSes, which are largely asymmetry agnostic. For this reason, RR has been widely used as a baseline for comparison [15], [24], [39]. Unfortunately, RR constitutes a suboptimal fairness solution [11], since it does not consider per-thread big-to-small speedups when distributing big-core cycles.

Currently, ACFS [11] constitutes the state-of-the-art OS-level fairness-aware scheduling scheme for AMPs. In [11] the authors experimentally demonstrated that ACFS clearly outperforms previous fairness-aware schedulers, such as RR [24], Equal-Progress [23], and A-DWRR [2], for a wide range of workloads running on real AMP hardware. To optimize fairness, ACFS leverages per-thread SF values to continuously track the relative progress that each thread in the workload makes on the AMP, and enforces fairness by evening out the slowdown observed across applications. The main limitation of ACFS [11] (also present in earlier schemes [2], [23]) is the fact that the scheduler does not take shared-resource contention effects into consideration. As our experiments reveal, failing to cater to these effects leads the scheduler to exhibit unfair behavior when multiple memory-intensive programs are included in the workload. CAMPS effectively improves fairness in this scenario.

To the best of our knowledge, the only existing contention-aware scheduling algorithms for AMPs are those proposed in [40] and [41]. However, unlike our OS-level approach, these schemes were designed as user-level scheduling prototypes that bind thread to cores via CPU affinity system calls. Fan et al. [40] present a scheduler for AMPs that strives to improve the system throughput when using workloads consisting of single-threaded programs. It relies on two prediction models *–specific to each application–* enabling the scheduler to approximate the degradation that the application suffers at runtime due to contention. Generating these prediction models (platform specific) for each application, requires to go through an offline training phase that entails running 80 workloads wherein the application is included. Our proposal –primarily designed to optimize fairness rather than throughput– does not rely on platform-specific or per-application prediction models, thus preventing the user from conducting the extensive offline profiling required to build those models [40]. Moreover, as opposed to our approach, [40] assumes that an application speedup factor is known beforehand (e.g., determined offline); this assumption is unrealistic on general-purpose systems. Barati et al. [41] propose a fairness-aware scheduler specifically tailored to asymmetric systems where cores differ in processor frequency only. It is well known that an application's degree of memory intensity is enough to approximate its slowdown when it runs on cores with the same microarchitecture but different frequency [11], [12]. For that reason, relying exclusively on the memory access rate (as in [41]) is effective under frequency-based asymmetry [10]. Notably, previous work [10], [13] has demonstrated that this form of performance asymmetry differs substantially from that of commercial AMP hardware available today, where the various cores exhibit profound microarchitectural differences and diverse cache sizes. In this scenario, other aspects beyond an application's degree of memory intensity must be taken into consideration for effective scheduling [10], [11], [14]. Unlike [41], our approach –implemented in the OS kernel rather than as a user-space scheduling prototype– does not make any assumption about the form of performance asymmetry of the platform. This enables us to perform an extensive comparison with recent fairness-aware approaches [11], [23] by employing real AMP hardware.

4 The CAMPS Scheduler

CAMPS consists of two components: the *performance monitor* and the *core scheduler*. The performance monitor gathers the value of various runtime metrics for each thread in the workload using performance counters, and feeds the core scheduler with critical information it needs, such as estimates of threads' slowdowns. The *core scheduler* assigns threads to big and small cores so as to preserve load balance, and swaps threads between cores when necessary to ensure that applications achieve similar progress on the AMP.

In this section we first present general aspects regarding our implementation of CAMPS in the Linux kernel. Then we outline the progress tracking mechanism and discuss how fairness is enforced via thread swaps. Next, we cover the non-work-conserving (NWC) mode of CAMPS, which may be triggered on special occasions to aid the performance monitor in approximating the slowdown of specific threads. Finally, we describe special features included in the scheduler to effectively deal with multithreaded applications. Notably, the description of the mechanism used by CAMPS to trade fairness for throughput and its evaluation can be found in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2018.2836418.

4.1 CAMPS in the Linux Kernel

The Linux scheduler is equipped with multiple scheduling algorithms (CFS, FIFO, etc.), which are implemented as independent *scheduling classes*. CAMPS's core scheduler was bundled as a new scheduling class in the kernel. In creating this class, we started off with a clone of CFS (fair class), and implemented CAMPS's *core scheduler* on top of it. By contrast, the *performance monitor* (platform specific) was implemented in a loadable kernel module –bundled as a monitoring module of the PMCTrack tool [42].

It is worth noting that Linux CFS is largely asymmetry agnostic; as we show in Section 5.2, CFS may randomly assign an application to different core types in subsequent runs of the same workload, which leads to inconsistent performance across executions on an AMP system. Moreover, CFS is contention unaware [43] and does not feature any mechanism to keep track of the progress that a thread makes as it runs on the different core types throughout the execution. (Actually, to CFS, a *tick* consumed on a big core *is worth the same* as a tick consumed on a small core [2].) As a result, and unlike CAMPS, CFS does not guarantee similar progress (fairness) across applications on AMPs.

Our scheduling class just relies on the stock Linux scheduler for two main tasks: (1) to enforce load balance between cores of the same type (big cores and small cores separately), and (2) to multiplex CPU usage among threads assigned to the same CPU (i.e., the CFS algorithm is applied on a per-CPU basis). CAMPS's core scheduler, by contrast, takes care of enforcing system-wide load balance, and evens out relative progress among applications, by assigning threads to the different core types and by triggering migrations if necessary. Since CAMPS is based on CFS, it maintains per-CPU run queues of runnable threads. In addition, it employs two linked lists of runnable threads, with threads assigned to big cores and to small cores, respectively; each list is protected with a read-write spinlock. Note that these lists are manipulated much less often than per-CPU run queues (e.g., when a thread is migrated onto a different core type). We observed that this design approach is not subject to scalability issues on off-the-shelf AMPs, as the ones we used, which feature a limited number of cores (up to 8). Notably, previous research [44] has demonstrated that even relying on a single global run queue delivers more than sufficient scalability on current AMP platforms. Nevertheless, to make CAMPS more scalable for future AMPs with a higher core count, the scheduler could be reimplemented by leveraging the core-partition approach described in [12].

4.2 Determining the Slowdown at Runtime

The *performance monitor* approximates a thread's current slowdown by using Eq. (2); the actual IPS is measured with

PMCs, and the IPS_{alone} is estimated by using a *history table* maintained for each thread at runtime. This table stores IPS values observed in past execution phases when the thread ran a big core in a low-contention scenario. As shown in Section 2, on a big-core cluster, the performance degradation that comes from interference with threads running on the small core cluster is typically very low. Based on this observation, big-core low-contention IPS values recorded in the table are used to approximate IPS_{alone}.

To detect low-contention scenarios on a big core, we leverage the heuristics based on the bus transfer rate metric proposed in [27], [45]. Essentially, a thread whose BTR is smaller than a given *low_btr* threshold is not likely to suffer noticeably from contention. In a similar vein, when the aggregate BTR in a core cluster falls below a given *high_btr* threshold the degradation due to contention is typically very low [27]. As shown in [27], [45], the thresholds can be easily determined for any platform by using synthetic benchmarks. Essentially when the thread runs on a big core in this kind of low-contention scenarios we assume that its slowdown is 1 (no degradation). If these scenarios do not occur naturally as a result of the contention-aware thread assignments performed by CAMPS, the core scheduler will enter a *non-work-conserving mode* (described in Section 4.4), which introduces low-contention scenarios artificially.

Indexing a thread's history table, which is necessary to approximate the slowdown and to record new IPS samples, requires the performance monitor to figure out whether information on the current execution phase already exists in the table or not. To this end, we leverage a variant of the phase-detection mechanism employed in previous work [46]. Overall, the scheduler continuously monitors the percentage of instructions of different types (int/FP, load, store, branches, etc.) retired during the last monitoring interval, which make up a *instruction type vector* (ITV). If the Manhattan distance of the ITVs for two performance samples (collected at different intervals) is smaller than a threshold, both samples are assumed to belong to the same execution phase. Note that the Manhattan distance of two n-dimensional vectors (X and Y) is defined as $\sum_{i=1}^{n} |X_i - Y_i|$.

Unfortunately, this phase-detection scheme, whose effectiveness was evaluated on a simulator [46], cannot be implemented in the real AMP platforms we used (presented in Section 2.2), as the performance monitoring unit is not equipped with the necessary performance events or with enough physical PMCs. To overcome this issue, we adapted the phase-detection approach by monitoring the thread's BTR and its IPS (required for our scheduling policy) along with two alternative *control metrics*: the number of L1 cache accesses per 1K instructions, and the percentage of branches retired over the total instruction count. As the ITV, the value of these control metrics for a specific phase remain the same under different levels of shared resource contention, and they do not vary significantly across core types. Notably, the value of these metrics changes dramatically when an application enters a new phase exhibiting a different degree of memory intensity and branch-prediction related behavior. These two aspects have a great impact on cross-core relative performance on AMPs [10], [11]. These observations make the selected *control metrics* very suitable to index the table.

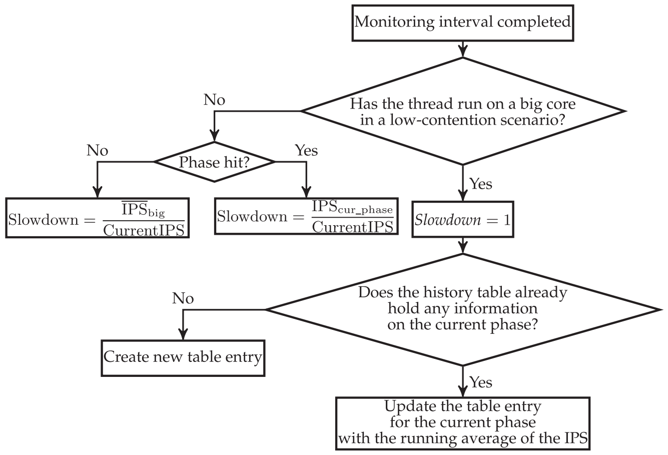
Monitoring interval completed
Has the thread run on a big core in a low-contention scenario?
No
Yes
Phase hit?
No
Yes
Slowdown = $\frac{\overline{\text{IPS}}_{\text{big}}}{\text{CurrentIPS}}$
Slowdown = $\frac{\text{IPS}_{\text{cur\_phase}}}{\text{CurrentIPS}}$
Slowdown = 1
Does the history table already hold any information on the current phase?
No
Yes
Create new table entry
Update the table entry for the current phase with the running average of the IPS

Fig. 3. Mechanism used by CAMPS for approximating a thread's slowdown with help from the history table.

Fig. 3 depicts how the *performance monitor* estimates a thread's slowdown and maintains its history table. The table is updated at the end of a monitoring interval in which the thread ran on a big core cluster in a low-contention scenario. If the table does not already hold information on the current phase, that IPS value is recorded in a new entry; otherwise the existing table entry is updated with a running average of the IPS values recorded for that phase. In either case, the scheduler estimates the slowdown to be 1 (no degradation). When the thread runs on a small core, or on a big core under potential contention, CAMPS accesses the history table to estimate the slowdown. If the IPS for the current phase is found in the table (i.e., *phase hit*), the slowdown is estimated with the ratio of the IPS value retrieved from the table ($\text{IPS}_{\text{cur\_phase}}$) and the current IPS value measured in the last monitoring interval. In case that no information is found for the current phase (i.e., *phase miss*), the estimated slowdown is the ratio of the average IPS across samples stored in the history table ($\overline{IPS}_{big}$) and the current IPS value.

To determine the most suitable size for the history table we conducted a sensitivity study by analyzing performance traces gathered with PMCs for SPEC CPU applications. This sensitivity study can be found in Appendix B, available in the online supplemental material. Based on the results of our analysis we opted to use history tables of 22 entries. This choice provides a good trade-off between slowdown estimation accuracy and memory utilization.

4.3 Progress Tracking and Enforcing Fairness

CAMPS's core scheduler maintains a progress counter for each thread referred to as `amp_progress`. This counter tracks how much progress the thread has made thus far relative to the progress that would have resulted from running it on a big core the whole time in complete isolation (no contention). When a thread runs for a clock *tick* on a given core type, the scheduler increments `amp_progress` by $\Delta_{\texttt{amp\_progress}}$, defined as follows:

$$\Delta_{\texttt{amp\_progress}} = \frac{100 \cdot W_{\text{def}}}{CS \cdot W_t}, \qquad (3)$$

where W_t is the thread's weight, derived directly from the application priority (set by the user); W_{def} is the weight of applications with the default priority; and CS is the thread's current slowdown as estimated by the *performance monitor*.

To illustrate the main idea behind the definition of $\Delta_{\mathtt{amp\_progress}}$, let us analyze the following example. A sequential program with the default priority (i.e., $W_{\mathrm{t}} = W_{\mathrm{def}}$) runs on an AMP system, and its single runnable thread is mapped to a big core. Suppose further that the thread is not currently suffering from contention. In this scenario, *CS* would be 1 (no degradation), so $\Delta_{\mathtt{amp\_progress}}$ would be equal to 100. This indicates that the thread is now making 100 percent of its maximum attainable progress, as it runs on a big core without contention. Let us now consider that the thread is eventually migrated onto a small core, where it experiences a relative slowdown (*CS*) of 2.5; this hypothetical slowdown comes from running on a less powerful core coupled with the potential degradation due to sharing resources with other threads. Under these circumstances, $\Delta_{\mathtt{amp\_progress}}$ would be equal to 40; namely, the thread only makes 40 percent of its maximum attainable progress (achieved when running on a big core in isolation). Therefore, in general, the lower the slowdown (*CS*), the faster a thread's `amp_progress` counter will be incremented.

When a new thread enters the system, the *core scheduler* assigns it to the least loaded core on the AMP, so that the load balance across cores is preserved. In doing so, CAMPS picks big cores first, since this contributes to maximizing throughput [2], [15]. Notably, the `amp_progress` counter of a newly created thread is set to the maximum value for this counter observed among threads in the system at that point. This initial value enables a *fair* progress comparison among threads that entered the system at different points in time. Every thread also has to go through a warm-up period (10 sampling intervals in our experimental setting) right after being spawned. The first two samples collected during the warm-up period are discarded for slowdown estimation, so as to mitigate mispredictions associated with cold-start effects (e.g., the number of cache misses typically spikes intermittently at the beginning of the execution). Moreover, throughout the entire warm-up period, a thread is not allowed to trigger the activation of CAMPS's NWC mode (described in Section 4.4). This is a control measure to remove the potentially negative interference caused by the presence of multiple short-lived memory-intensive threads, which could otherwise activate the NWC mode ineffectively; that would lead to unnecessary overheads (due to potential core disabling actions) without reaping any benefits, as CAMPS discards a thread's history table when it terminates.

Note that the approach used by CAMPS to enforce fairness via progress tracking has several aspects in common with that of the ACFS scheme [11]. Despite the fact that both schedulers maintain per-thread progress counters, they employ different mechanisms to determine a thread's current slowdown (denoted as the *CS* factor in Eq. (3)) at runtime. While CAMPS does take shared resource contention into consideration, as described in Section 4.2, ACFS does not. In fact, ACFS assumes that a thread's slowdown is always 1 when it runs on a big core, and uses the thread SF (predicted via a platform-specific estimation model) to approximate its slowdown when the thread runs on a small core.

Like ACFS, CAMPS may also trigger thread swaps between cores every so often to enforce fairness. Essentially, threads mapped to big cores usually make faster progress than threads running on small ones, which causes unfairness. To even out the progress among threads via thread swaps, CAMPS follows a similar approach to that of ACFS [11]. Specifically, a thread running on a big core will be swapped with another thread running on a small core only when the difference of their progress counters exceeds a given threshold, referred to as `amp_threshold`. Specific instructions are provided in [11] for selecting the most appropriate value of this threshold for a given platform. For our experiments, we chose a value of this threshold so as to achieve an average migration rate of 400 ms, which ensures negligible overheads in current AMP hardware [11].

It is worth highlighting that special care is taken with sleeper threads (i.e., those that wake up after a potentially long suspension). Essentially, the progress counter of a sleeper thread that just woke up could be much smaller than that of other threads in the system, as the thread's progress counter remains unmodified while it sleeps. This situation could lead sleeper lagging threads to monopolize big cores when waking up after a very long pause. To address this issue, CAMPS resets a thread's progress counter when it realizes that it has been blocked for a certain time period, which is application specific. This period corresponds to the time that it would take this thread when just migrated to a small core (due to a fairness-oriented swap) to be swapped back to a big core. This time period depends on `amp_threshold` and on the thread's average slowdown, which is maintained by CAMPS. The reset value for the counter is the minimum value for the progress counter observed among threads on the system.

We found that relying on the progress counters alone (as ACFS does) is ineffective in case that aggressor applications and contention-sensitive programs are often mapped to the big-core cluster simultaneously. As shown in Section 2, this mapping may severely degrade the performance of contention-sensitive applications, which may backfire by decreasing the benefits from using a big core. To mitigate this issue, CAMPS uses the BTR-based heuristics proposed in [27] to detect potentially contentious scenarios, and favors those threads swaps that contribute to avoiding contention on the big core cluster. Algorithm 1 illustrates how CAMPS's core scheduler selects threads to be swapped. The algorithm is executed as soon as the scheduler detects that swap candidates exist on both core types (i.e., the progress counters of two threads running on opposite core types exceed `amp_threshold`). Specifically, CAMPS always selects the thread with the highest `amp_progress` counter running on a big core –denoted as T_B– to be migrated to a small core. In choosing its swap partner, small-core threads with a lower value of the `amp_progress` counter are considered first. If a contention-friendly swap is found (i.e., it leads to a low contention scenario on the big core cluster), the swap is performed. Otherwise, the thread with the lowest BTR is the one selected as the swap partner; this contributes to reducing the degree of shared-resource contention on the big core cluster as a result of the reduction in the cluster's aggregate BTR [27]. Note also that the scheduler forces the selection as a swap candidate of those threads that are lagging considerably behind the rest, namely, when the difference between T_B's progress counter and the thread's progress counter is greater than $2*$`amp_threshold`. This enables aggressor (high-BTR) threads to eventually have a chance to run on big cores when the workload includes multiple memory-intensive applications.

Algorithm 1. Selection of Swap Candidates in CAMPS

```
Input: $T_B$ is the runnable thread with the highest progress
       counter mapped to the big core, $S$ is the set of runnable
       threads $(T_S, i)$ assigned to small cores that constitute
       potential swap partners for $T_B$ (i.e., amp_progress
       $(T_B)$ $-$ amp_progress$(T_S, i)$ $\geq$ amp_thresh). Note
       that $S \neq \emptyset$, and threads in $S$ are sorted in ascending
       order by their amp_progress counter.
  $min\_btr \leftarrow \infty$; $T_{min-BTR} \leftarrow$ $NIL$;
  $swap\_performed \leftarrow$ false;
  do
    $T_S, i \leftarrow$ Get first thread in $S$ ;
    if Swapping $T_B$ and $T_S, i$ leads to a low-contention scenario
    on the big core cluster || (amp_progress$(T_B)$ $-$ amp_
    progress$(T_S, i) \geq 2*$amp_threshold) then
      Swap $T_B$ and $T_S$;
      $swap\_performed \leftarrow$ true;
    else
      Remove $T_S, i$ from $S$;
      If $BTR(T_S, i) < min\_btr$ then
        $min\_btr \leftarrow BTR(T_S, i)$;$T_{min\text{-}BTR} \leftarrow T_S, i$ ;
      end
    end
  while !$swap\_performed$ && $S \neq \emptyset$
  if !$swap\_performed$ then
    Swap $T_B$ and $T_{min\text{-}BTR}$;
  end
```

4.4 Non-Work Conserving Mode

As discussed earlier, CAMPS populates a thread's history table while it runs on a big core cluster during low contention scenarios. Unfortunately, when the number of memory-intensive threads in the workload is high, low contention scenarios might not occur that often for contention-sensitive programs. In these cases, CAMPS may transition into a *non-work-conserving mode*, in which low contention scenarios are created artificially. To control transitions into this special mode, CAMPS operates as follows. Every time that a thread completes k consecutive monitoring intervals (being k configurable), the scheduler retrieves the thread's phase hit rate as well as the number of IPS samples that have been inserted into the history table over that time period. If the phase-hit rate falls below 80 percent, and no IPS samples have been inserted in the history table during that period, the scheduler enters the NWC mode. We will refer to the thread that caused the transition into this mode as the *NWC thread*.

When in the NWC mode, fairness-oriented thread swaps are not performed. During this special mode, the main goal is to collect as many low contention big-core IPS samples as possible for the *NWC thread*. To this end, if the *NWC thread* was not running on a big core already, it will be swapped with a big-core thread. In doing so, CAMPS tries to select a memory-intensive (high-BTR) thread as the swap partner, so as to reduce contention on the big core cluster as a result of the swap. Once the NWC thread is mapped to the big core, CAMPS will attempt to gather big-core IPS samples for this thread. If at this point a low-contention scenario does not yet occur on the big core cluster, the scheduler will temporarily disable (for a very short period of time) as many big cores as necessary to create such a scenario. In practice, making this possible comes down to disabling only a few big cores: those where memory-intensive threads are currently running. Note that during the NWC mode, other threads (in addition to the NWC thread) may leverage low-contention scenarios to populate the history table.

The scheduler will transition back into the normal operating mode when (1) the NWC thread's phase hit rate is over 80 percent –after inserting a number of IPS samples in the history table–, or (2) when the NWC thread blocks or terminates. Notably, when in the NWC mode, CAMPS still keeps updating thread progress counters. This allows threads that did not benefit the NWC mode (e.g., those assigned to big cores that were temporarily disabled), to be compensated later accordingly. In addition, to prevent that specific threads force the transition into the NWC mode systematically, we take progress counters into consideration when controlling transitions; threads progressing much further ahead than the rest at some point cannot become NWC threads.

In our implementation in the Linux kernel, big cores are temporarily disabled in the NWC mode (when needed) by selecting the *idle task* to run forcefully on the corresponding core (this action is performed in the `pick_next_task()` operation of our scheduling class), and by temporarily binding to that core any thread previously assigned to it. We found that using short core disabling periods, such as the 100 ms setting used in our experimental platforms (2 monitoring intervals), allows the scheduler to have a fine-grained control when in the NWC mode. Essentially, this enables CAMPS to better adjust to the number of core disabling operations required by the current NWC thread.

Although CAMPS was designed primarily for long-running compute-intensive workloads, latency-sensitive memory-intensive applications could be negatively affected by core disabling actions in the NWC mode. To deliver more consistent tail latencies, CAMPS could be seamlessly modified to map this kind of applications to small cores, which are never disabled when in the NWC mode.

4.5 Special Support for Multithreaded Applications

On AMPs, an application can be developed so as to explicitly leverage the features of the various cores by dividing the computation into multiple tasks or threads specifically designed to run effectively on a particular core type. These applications are typically run by manually binding the various threads/tasks to the core type where they are meant to run. CAMPS supports the execution of those applications, as it respects user-enforced CPU affinities. Nevertheless running such an application along with other programs would not guarantee system-wide fairness; CAMPS strives to deliver fairness only across those (unmodified) applications whose threads are allowed to run on different core types. Notably, affinities in general greatly limit the schedulability in most OS-level schedulers [47], not only that of CAMPS.

To provide better support for multithreaded programs that do not rely on affinities, CAMPS leverages two mechanisms: *spin notifications* and *per-application history tables*.

Spin notifications enable the scheduler to be aware of those situations where threads in a multithreaded program busy wait (or *spin*) rather than blocking while waiting in synchronization primitives, such as barriers. Busy waiting enables to substantially reduce the number of context switches performed by the OS scheduler [48], and it may also reduce the number of thread migrations on AMP systems [15]. Nevertheless, *spinning threads* must be properly handled by the scheduler. The main issue is that busy-waiting threads may achieve a high IPS despite not doing useful work. (Best practices in implementing spin locks dictate using algorithms where a thread spins on a local variable [48]; this leads to a high IPC, due to the effective utilization of the CPU pipeline.) Using these misleading IPS values under CAMPS, would lead to polluting the history table and, in turn, to serious slowdown mispredictions.

To address this issue, we leverage *spin notifications* from user space to the OS by using a variant of the technique proposed in previous work [15]. In our implementation, we maintain a memory region shared between each application thread and the OS. When a thread begins to spin, it activates a flag in the shared memory region, which is later disabled as soon at the thread stops spinning. We opted to use a shared memory region rather than system calls (as in [15]) for spin notifications, since the former approach provides negligible overhead. When a thread is spinning, the *performance monitor* discards the associated IPS samples, and always estimates the slowdown for the thread to be 1, as it is not doing useful work. In a similar vein, CAMPS's *core scheduler* avoids migrating spinning threads to big cores. Notably, issuing spin notifications from user space does not require making changes in the applications as long as they use the synchronization primitives provided by the threading library or the underlying runtime system. As a proof of concept we implemented this mechanism in the OpenMP runtime system provided by GCC (bundled in a dynamic library), by instrumenting the code of synchronization primitives. For applications that do not use standard, library-based synchronization primitives, spin notifications could be exploited by leveraging hardware-aided spin-detection approaches [48] or by manually instrumenting the code.

In many multithreaded applications, the various threads do the same kind of processing but with different data. In this scenario, we can leverage the IPS samples stored in a thread's history table to aid in predicting the slowdown for the remaining threads in the application.To this end, we maintain two levels of history tables for multithreaded applications: the per-thread table (L1) –presented in Section 4.2, and a per-application history table (L2). Essentially, when a thread creates a new phase entry in its own table, it inserts this new entry into the per-application table too. In doing so, other threads in the application that incur a L1 phase miss, can potentially retrieve information for the current phase by accessing the L2 (application-wide) table. If a L2 phase hit occurs, the entry is copied onto the thread's private table (L1). This way we avoid future accesses to the same L2 table entry. Note that in limiting the number of read and write operations on the L2 table, we reduce potential contention that comes from accessing the L2 table (protected with a lock) simultaneously from multiple CPUs. Although using two levels of history tables is specially well suited to applications where all threads run the same code with different data, the scheme could be trivially augmented to other kind of multithreaded programs (such as those following the pipeline paradigm) where a few threads perform a specific task cooperatively, where others do a different kind of processing. In that case, a L2 history table would be shared by threads that do the same kind of processing, which could be identified by the function that they execute.

Lastly, we should highlight that, for multithreaded applications, CAMPS downscales the CS factor in Eq. (3) in proportion to the number of runnable threads in the application (a proxy for the amount of thread-level parallelism), like ACFS does[11]. Previous work [11], [13] has demonstrated that, when multithreaded programs are included in the workload, this approach enables the scheduler to provide better performance and fairness than making decisions based exclusively on per-thread slowdowns (or SFs).

5 Experimental Evaluation

In this section, we begin by comparing the effectiveness of CAMPS with that of previously proposed asymmetry-aware schedulers [10], [11], [23], [24], which, as CAMPS, primarily target long-running compute-intensive applications. The schedulers (evaluated in Section 5.1) were implemented as a scheduling class in the Linux kernel v3.10.104. By the time we started with the implementation, that was the latest stable kernel version with official manufacturer support for the Odroid XU4 board. Variants of the vanilla v3.10.104 kernel were used on the other AMP systems considered, to maintain a common scheduler code base.

In Section 5.2 we compare the degree of fairness and other aspects of our scheduling proposal with that of the stock Linux scheduler (CFS) and with its extension for ARM big. LITTLE platforms (HMP [33]). To this end we experimented with a broad spectrum of workloads (long and short-running CPU-bound programs, IO-intensive and latency-sensitive benchmarks, etc.). These additional experiments reveal that CAMPS is able to acceptably deal with various application types for which it was not optimized. In addition, the results illustrate the high variability delivered by CFS and HMP for long-running compute-intensive workloads, making both schedulers unsuitable baselines for comparison when using this kind of workloads on AMPs.

5.1 CAMPS versus Other Asymmetry-Aware Schedulers

To assess the effectiveness of CAMPS we compared it with three previously-proposed fairness-aware schedulers for AMPs: ACFS [11], Equal-Progress [23] and an asymmetry-aware Round-Robin (RR) scheme [24]. We also experimented with a scheduler that attempts to optimize throughput by preferentially running on big cores those applications that derive a higher big-to-small speedup [10], [13]. We will refer to this scheduler as *HSP* (High SPeedup).

All the schedulers considered (except for RR) rely on performance monitoring counters to function. HSP and ACFS determine threads' SFs on-line by continuously monitoring different PMC events, and by feeding an estimation model with the obtained event counts. (More information on the mechanism employed to build the estimation model and to determine the associated events on our ARM-based experimental platforms can be found in [38].) The Equal-Progress scheduler [23], by contrast, leverages PIE [14] or IPC

TABLE 1
Multi-Application Workloads for the 2B-4S AMP Configuration

Name	Applications	Name	Applications
W1	GemsFDTD,equake,soplex,milc,ammp,bzip2	W13	GemsFDTD,bwaves,gamess,hmmer,crafty,astar
W2	galgel,soplex,hmmer,lbm,fma3d,bzip2	W14	bzip2,bwaves,hmmer,lucas,gobmk,gzip
W3	galgel,equake,gamess,lbm,bzip2,astar	W15	soplex,art,vortex,lbm,fma3d,gobmk
W4	twolf,bwaves,equake,soplex,astar,gobmk	W16	galgel,equake,hmmer,lbm,fma3d,h264ref
W5	GemsFDTD,bwaves,equake,povray,fma3d,astar	W17	bwaves,equake,gamess,povray,astar,libquantum
W6	bwaves,equake,gamess,lbm,fma3d,bzip2	W18	GemsFDTD,galgel,gamess,hmmer,astar,libquantum
W7	GemsFDTD,applu,perlbmk,sixtrack,astar,gzip	W19	swim,mcf,perlbench,h264ref,gobmk,gzip
W8	bwaves,perlbmk,povray,fma3d,astar,gzip	W20	galgel,equake,hmmer,povray,mgrid,gobmk
W9	galgel,perlbmk,sixtrack,mgrid,astar,libquantum	W21	galgel,equake,hmmer,bzip2,perlbench,h264ref
W10	GemsFDTD,vortex,perlbmk,fma3d,astar,gzip	W22	galgel,equake,gamess,hmmer,sixtrack,povray
W11	bzip2,equake,hmmer,vortex,crafty,astar	W23	gamess,art,bzip2,gobmk,sixtrack,vortex
W12	gamess,hmmer,soplex,art,astar,gzip	W24	galgel,gamess,hmmer,povray,perlbench,gobmk

sampling [24] to determine thread SFs at runtime. Since the required hardware extensions for PIE are not available in commercial AMP hardware, we evaluate the history-based variant of Equal-Progress [23], based on IPC sampling. Under all schedulers, PMCs are sampled each 50ms on a per-thread basis; this sampling period enables the OS to detect coarse-grained program phases and to filter out many spikes in performance metrics that become apparent when using smaller sampling periods (due to fast oscillations in some metrics). Notably, we observed that the overhead associated with PMC-related processing at this rate becomes negligible for most applications (for a few programs we observed up to a 0.28 percent overhead). To reach a 1 percent overhead, the sampling period has to be reduced to a value as low as 5 ms.

For our experiments, we used the 2B-4S and 4B-4S AMP configurations (ARM big.LITTLE based) presented in Section 2.2. Our evaluation targets workloads consisting of long-running compute-intensive benchmarks from diverse suites (SPEC CPU, PARSEC, Minebench and NAS Parallel). We also experimented with `FFTW3D`—a program performing the FFT. All programs were compiled with GCC (`-O3` switch) and by employing the `-mtune=cortex-a15.cortex-a7` (4B-4S only) and the `-mtune=cortex-a57.cortex-a53` (2B-4S only) compiler options to apply common big.LITTLE optimizations. The total thread count in each workload was set to match the total number of cores in the platform (including both big and little cores), as in previous work on AMPs [10], [13], [23]. We ensure that all applications in the mix are started simultaneously and when one of them terminates it is restarted repeatedly until the longest application in the set completes three times. We then measure unfairness and throughput, by using the geometric mean of the completion times for each program. To assess throughput we employed the *Aggregate Speedup* (ASP) metric as in [11], [38]. We ran each experiment five times, and report the average, minimum and maximum values of the unfairness and throughput in each case.

In evaluating the various schedulers we built two different sets of workloads, shown in Tables 1 and 3. In the first one, each program mix is made up of six single-threaded applications running on the 2B-4S configuration. The second set, which we ran on 4B-4S, includes mixes consisting of both single-threaded and multithreaded programs.

5.1.1 Workloads for the 2B-4S Configuration

We begin by analyzing the results of the first workload set, shown in Fig. 4. The unfairness and throughput (ASP) values reported in the charts are normalized with respect to the results of the HSP scheduler. In building the workloads (Table 1), we divided SPEC CPU applications into two groups: *light-sharing* programs, whose performance do not suffer noticeably under contention; and *memory-intensive* programs, which are subject to high contention-related

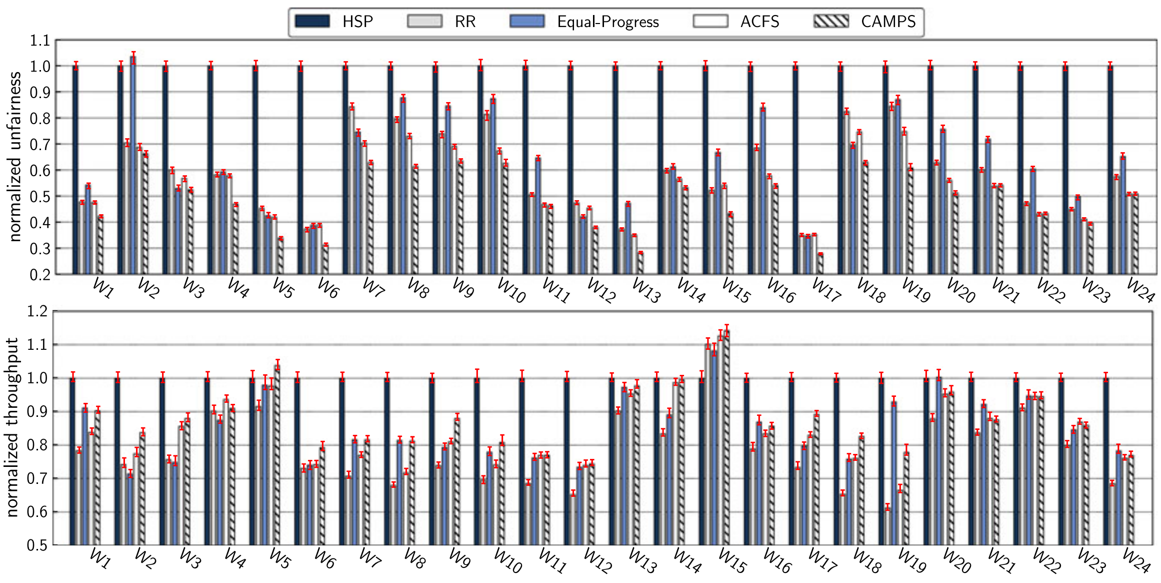

Fig. 4. Unfairness (top) and throughput (bottom) for the workloads in Table 1 running on 2B-4S under the various scheduling algorithms.

TABLE 2
Average Reduction in Unfairness and Increase in Throughput Achieved by CAMPS Over the Other Schemes on the ARM Juno Board

CAMPS vs. others	Reduct. in Unf.	Increase in throughput
HSP	50.96%	−12.24%
RR	17.08%	13.19%
Equal-Progress	23.64%	3.31%
ACFS	10.71%	4.48%

degradation or put significant pressure on the shared resources. We then generated 24 random program mixes by combining 29 SPEC benchmarks that cover a wide spectrum of speedup factors. Table 1 shows these program mixes which are displayed sorted in descending order by the number of memory-intensive programs included in the workload.

The results illustrate that optimizing one metric may lead to substantial degradation of the other one. This trend was also observed in previous work [11], [38], which illustrates that fairness and throughput are largely conflicting optimization goals on AMPs. As is evident, HSP, which strives to optimize throughput, achieves the best ASP values for most workloads, at the expense of the worst unfairness numbers (the higher, the worse) across the board. Conversely, the remaining schedulers (fairness aware), achieve substantial reductions in unfairness versus HSP (up to a 72 percent reduction—CAMPS under W17), at the cost of potentially high throughput degradation (up to 38 percent—RR under W19).

The results of ACFS, RR and CAMPS exhibit a clear trend across the board. Specifically, for the vast majority of workloads ACFS delivers better throughput and higher reductions in unfairness than RR. This is the expected behavior since ACFS takes applications' big-to-small speedups into consideration when distributing big-core cycles among applications, whereas RR does not. Despite the higher throughput, the fact that ACFS does not factor in contention effects when making decisions, leads ACFS to similar unfairness figures to those of RR in some cases (e.g., W4-W6, W15 or W17). By contrast, our proposal is able to reduce unfairness even further: by up to 11 percent with respect to ACFS (W17) and by up to 28 percent relative to RR (W19). In addition, CAMPS is capable of reaping higher throughput gains: up to a 17 percent increase versus ACFS (W19). Notably, under those workloads including a small number of memory-intensive applications (W20-W24), we observe that CAMPS and ACFS perform very similarly. This suggests that CAMPS is also suitable for low-contention scenarios, as it delivers similar unfairness and throughput figures to ACFS, the state-of-the-art fairness-aware scheme providing the best results under these circumstances [11]. All in all, as summarized in Table 2 CAMPS achieves an average 10.7 percent reduction in unfairness with respect to ACFS while improving throughput by 4.48 percent.

Now we zoom in on the results of the Equal-Progress scheme, which, as our proposal, also strives to optimize fairness. We observe that this scheduler is not able to obtain lower unfairness than CAMPS or ACFS for most workloads. More importantly, Equal-Progress's results reveal significant divergences across the board: for a few workloads, such as W4, W12 or W18, it obtains throughput and fairness figures closer to those of CAMPS and ACFS, whereas for others it exhibits a much unfairer behavior along with either throughput degradation relative to CAMPS (e.g., W2, W8-W10, W14, etc.) or with throughput gains in some cases (e.g., W1, W19 or W20). As discussed in detail in [11], this somewhat inconsistent behavior of Equal-Progress stems from two main factors: (1) the inaccuracies associated with the mechanism it employs to track thread progress on AMPs, and (2) the fact that it relies on IPC sampling to determine thread's SFs online on commercial AMPs [23]. IPC sampling has been shown to lead to inaccurate SFs, since IPC values collected on each core type may belong to different program phases [12]. We observed that inaccuracies in the SF –obtained when measuring the IPC directly on both core types– are more frequent under contention, as the IPC may suffer profound oscillations (even within the same program phase) based on the degree of contention a thread is suffering. Inaccuracies prevent Equal-Progress from delivering even progress across applications, and the performance it delivers is heavily affected by these inaccuracies: throughput increases when the scheduler happens to grant a higher big-core share to high-speedup applications. Although CAMPS also relies on measuring the IPC to determine a thread's slowdown, the reference values used to approximate run-alone performance (IPS_{alone}, stored in the history table for the different phases) are collected under low contention scenarios, as explained in Section 4. This makes it possible for CAMPS to overcome the aforementioned issue of Equal-Progress. On average, our proposal reduces unfairness by 23.6 percent compared to Equal-Progress.

The results also reveal that for some workloads CAMPS and ACFS achieve throughput values similar (in a 3 percent range) to those of HSP. Overall, we observe that the throughput degradation achieved by fairness-aware schedulers is significantly lower for workloads where the number of applications that experience a higher-than-average speedup exceeds the number of big cores (two). Under these circumstances (e.g., W1, W3, W13 or W17), CAMPS and ACFS grant a substantial amount of big core cycles to these specific applications (by triggering periodic swaps), whereas HSP usually maps only two high-speedup programs to big cores for a long time period. This leads ACFS and CAMPS to reduce unfairness in a greater extent than HSP (e.g., W13 and W5), while yielding a low throughput degradation.

Lastly, we should highlight that HSP is especially affected by contention effects under the W5 and W13-W15 workloads, where the two applications with the highest speedup (those listed at the beginning of each row in Table 1) are both highly memory intensive or constitute a pair consisting of a memory-intensive and a cache-contention sensitive program. The benefit that these applications derive from running on a big core comes in part due to the fact that this core type features a larger shared L2 cache than the small core. Unfortunately, when the scheduler maps two memory-bound programs on the big cores simultaneously, threads compete with each other for space in the shared cache as well as for bus bandwidth, which leads to non-negligible performance degradation for both applications, and in turn degrades system throughput. Specifically, under the aforementioned workloads, HSP maps memory-bound applications to big cores simultaneously for longer periods of time than fairness-aware schedulers, which –by contrast– swap threads

TABLE 3
Multi-Application Workloads for the 4B-4S AMP Configuration

Name	Applications	Name	Applications
M1	`art,galgel,libquantum,sixtrack,gamess,hmmer,soplex,gzip`	M9	`art,libquantum,sixtrack,h264ref,semphy(4)`
M2	`galgel,libquantum,hmmer,mcf,mgrid,crafty,parser,gzip`	M10	`gamess,applu,povray,crafty,swaptions(4)`
M3	`mcf,lucas, galgel, soplex,h264ref,povray,perlbmk,gobmk`	M11	`soplex,povray,namd,gobmk,semphy(4)`
M4	`galgel,mcf,h264ref,povray,perlbmk,crafty,gobmk,astar`	M12	`fma3d,h264ref,povray,equake,kmeans(4)`
M5	`gamess,hmmer,mcf,mgrid,lucas,applu,namd,gobmk`	M13	`FFTW3D(4),kmeans(4)`
M6	`art,galgel,gamess,hmmer,mgrid,lucas,h264ref,astar`	M14	`semphy(4),EP(4)`
M7	`hmmer,mcf,mgrid,lucas,soplex,applu,h264ref,gzip`	M15	`blackscholes(4),EP(4)`
M8	`mgrid,lucas,soplex,fma3d,applu,ammp,h264ref,astar`	M16	`blackscholes(4),kmeans(4)`

between core types every so often. Swapping threads reduces the amount of time that the conflicting applications are mapped together to big cores, which contributes to improving both throughput and fairness. Specifically, the results reveal that all fairness-aware schedulers reap high normalized throughput figures under these program mixes (W5, W13-W15). More importantly, our proposal, is able to outperform HSP for some of these conflicting workloads (W5 and W15). This is possible thanks to the fact that CAMPS swaps threads based on their observed progress and by catering to the degree of contention.

5.1.2 Workloads for the 4B-4S Configuration

We now proceed with the discussion of the results for workloads we ran on the 4B-4S configuration. On this system, we attempted to analyze workload scenarios with a wide diversity of SFs among applications and a varying degree of competition for the available big cores. Note that, in building the program mixes, we had to pay special attention to the aggregate memory footprint of the workload, which should not exceed the limited amount of physical memory available on the Odroid XU4 board (2 GB) to prevent Linux's Out-of-Memory killer from kicking in during the experiments. Due to this constraint, we had to discard some application mixes for the different categories considered.

Overall, the workloads we explored –shown in Table 3– can be grouped in three broad categories. The first one combines 8 single-threaded programs (M1-M8) that exhibit a varying degree of memory intensity and cover a wide spectrum of SF values. Workloads in the second category (M9-M12) couple 4 single-threaded applications with a parallel program. Notably, the sequential programs derive a higher benefit from using a single big core most of the time than the multithreaded program. Catering to the amount of TLP (thread-level parallelism) in the application under these circumstances is crucial to identify those application phases that really benefit from using a handful of big cores (e.g., serial execution phases) [13], [15]. Finally, workloads in the third category (M13-M16) combine two parallel applications with different scalability features. Specifically, the `FFTW3D`, `semphy` and `blackscholes` programs exhibit sequential phases that span over 20 percent of their execution time, whereas `EP` and `kmeans` constitute highly parallel applications.

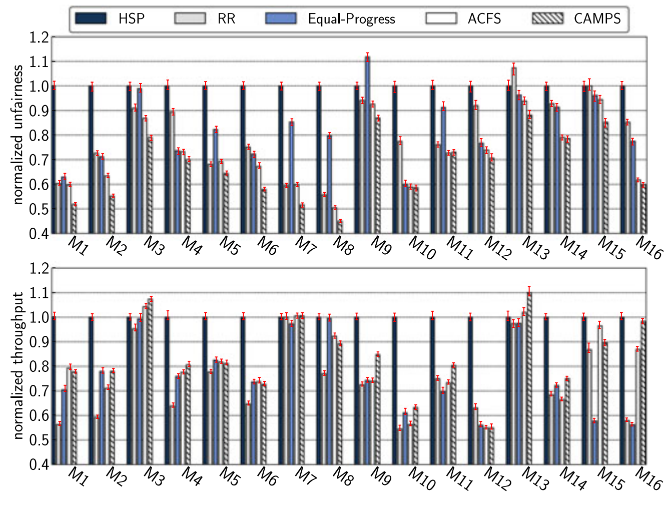

Fig. 5. Unfairness (top) and throughput (bottom) for the workloads in Table 3 running on 4B-4S under the various scheduling algorithms.

Fig. 5 shows the results for workloads in Table 3. Despite the profound differences between the composition of these workloads and those evaluated on the 2B-4S configuration, the results exhibit very similar trends to those discussed earlier. Essentially, CAMPS achieves the highest reduction in unfairness (up to 55 percent versus HSP) for the vast majority of workloads. At the same time, ACFS is usually the scheme that provides closest fairness figures to those of CAMPS, followed by RR and Equal-Progress. Again, we observe that ensuring fairness comes at the expense of significant throughput degradation in some cases (up to 45 percent - M12).

Results in Table 4 indicate that CAMPS still achieves substantial average reductions in unfairness w.r.t. the other schemes on 4B-4S (32.7 percent w.r.t. HSP, and 7 percent relative to ACFS). These overall gains are slightly smaller than those achieved on 2B-4S (see Table 2). This has to do with the lower degree of memory intensity of the workloads we ran on 4B-4S, which stems from the impossibility (due to the memory constraints) to consider mixes with multiple highly memory-intensive programs with a large memory footprint.

In spite of obtaining more modest fairness improvements in this scenario, CAMPS reaps considerably higher throughput gains relative to RR and Equal-Progress –over 16 percent and 12 percent respectively. This is due to the higher speedup diversity present in these program mixes, which stems from two factors. First, the SF range across sequential programs is significantly wider on this platform (from 1.36x

TABLE 4
Average Reduction in Unfairness and Increase in Throughput Achieved by CAMPS Over the Other Schemes on the Odroid XU4 Board

CAMPS vs. others	Reduct. in Unf.	Increase in throughput
HSP	32.75%	−16.03%
RR	16.89%	16.53%
Equal-Progress	18.67%	12.05%
ACFS	7.17%	4.51%

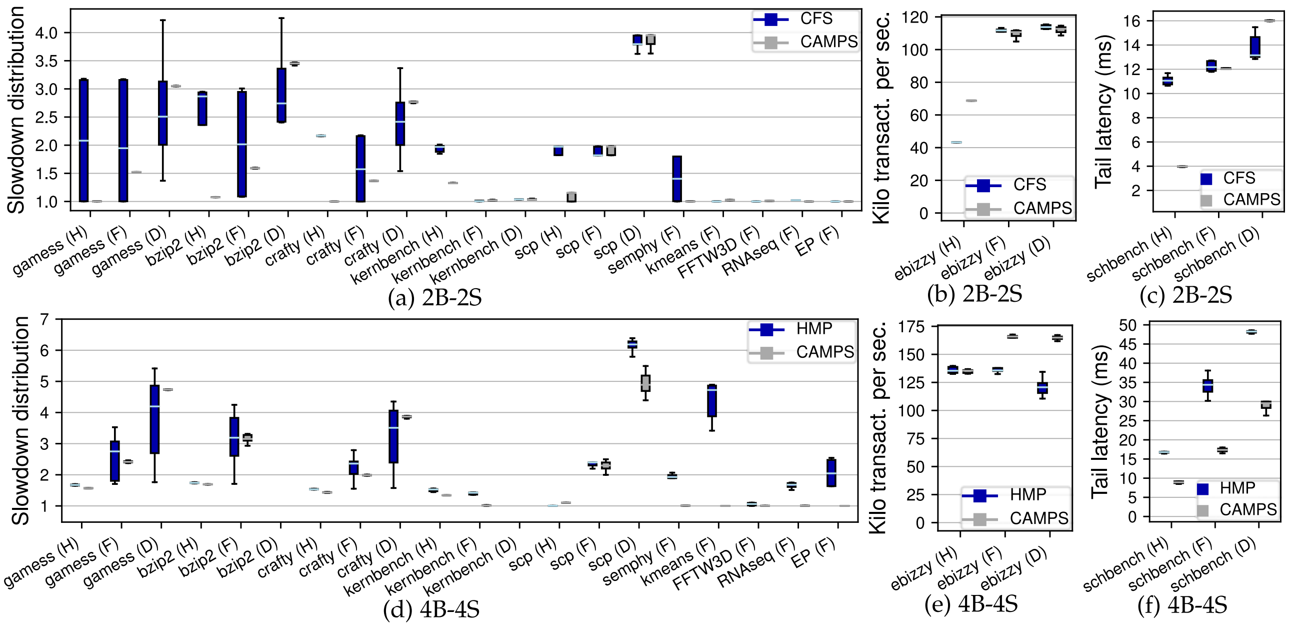

Fig. 6. CFS versus CAMPS on the Intel QuickIA –Figs. (a)-(c)–, and HMP versus CAMPS on the Odroid 4 XU board –Figs. (d)-(f)–.

to 6.63x) than on 2B-4S (from 1.5x to 4.4x). RR does not take SFs into consideration when making scheduling decisions, thus failing to obtain decent throughput figures in this context. Second, some program mixes combine single-threaded programs, which derive non-negligible benefits from using a single big core, with multi-threaded programs that only derive significant benefits from big cores in the event that all of its active threads are mapped simultaneously to big cores for some time (due to synchronization). Under these circumstances, devoting big cores to run low-TLP phases (e.g., serial code) brings higher benefits than mapping threads-to-cores based on the per-thread slowdown [11], [15], [34]. Unlike RR and Equal-Progress, the other schedulers (including CAMPS) take this aspect into consideration indirectly by downscaling the thread's slowdown factor (or speedup [13]) with the number of runnable threads in the application (a proxy for the amount of TLP), as stated in Section 4.5. Failing to cater to the amount of TLP in the application leads RR and Equal-Progress to high throughput degradation in some cases (e.g., over 40 percent degradation under M16).

5.2 CAMPS versus CFS and HMP

We now illustrate how CAMPS compares with the stock Linux scheduler (*CFS*) and with *HMP* [33] in terms of performance variability across runs. In doing so, we consider diverse workloads, beyond those which CAMPS was optimized for. For the comparison against HMP, we employed the Odroid XU4 board (4B-4S), as the kernel provided by the board's manufacturer uses HMP. To experiment with CFS, we used the Intel QuickIA (2B-2S); CFS is the default scheduler on this platform (to the best of our knowledge, no implementation for HMP is available for this x86 system).

To measure how an application's completion time varies across multiple runs of the same workload, we used different types of programs: long-running compute-intensive sequential applications (`gamess`, `bzip2` and `crafty`); a kernel compilation benchmark (`kernbench` [49]) that creates a mix of short-lived compute- and I/O-intensive single-threaded processes; a sequential I/O intensive benchmark (`scp`) that transfers a 200 MB file over the local network; and several multithreaded HPC programs (`semphy`, `kmeans`, `RNAseq`, `FFTW3D` and `EP`) with different scalability and synchronization patterns. We also experimented with `ebizzy` [49] –a micro-benchmark designed to generate a web server like workload (performance reported in terms of transactions per second)–, and with `schbench` –a benchmark [50] that measures the scheduler's tail (p99) latency.

In running these applications, we considered three homogeneous workload scenarios: *Full* or F –total thread count (N_T) matches the number of CPUs (N_{CPUS})–, *Half* or H ($N_T = N_{CPUS}/2$), and *Double* or D ($N_T = 2 \cdot N_{CPUS}$). For single-threaded programs we launched multiple simultaneous program instances to match the total thread count desired. Notably, for HPC compute-intensive parallel workloads, which are typically run with a total thread count that matches the number of CPUs, we experimented with Full Load only.

Figs. 6a and 6d show the slowdown distribution of the various workloads under CAMPS, CFS and HMP on 2B-2S and 4B-4S, respectively. For each workload, we ran the program at least 20 times so as to capture the variance of the performance distribution under each scheduler. The slowdown is normalized with respect to the fastest run registered when running the program alone on the system.

We begin by discussing the results of the CFS scheduler (Fig. 6a). In the H scenario, CFS provides worse performance (higher slowdown) and significantly higher variability than CAMPS across the board. This stems from the fact that CFS is asymmetry agnostic, and it randomly maps threads to any of the idle cores regardless of its type. Moreover, CFS tries to keep a thread running on the same CPU for as long as possible (to reduce the number of migrations), even if the thread is mapped to a small core. By contrast, CAMPS maximizes big core utilization so it maps all threads to big cores in this case ($N_T = N_{BigCores}$). As a result, it optimizes performance and evens out the slowdown (fairness).

In the F and D scenarios, both CFS and CAMPS evenly distribute threads across cores. The results, however, largely depend on the nature of the benchmark. For long-running compute-intensive programs CAMPS delivers repeatable completion times and similar unfairness values across runs, thanks in part to fairness-oriented thread swaps. By contrast, CFS exhibits a huge slowdown variability here; for instance, under the gamess (D) workload the slowdown ranges between 1.37x and 4.22x. That is because CFS does not make any effort to guarantee that threads make equal progress on an AMP, as discussed in Section 4.1. In fact, CFS may map an application to a big core in a whole run, and to a small core in another run. Our overarching conclusion is that for long-running compute-intensive workloads (like those considered in Section 5.1), the stock Linux scheduler does not constitute a good baseline for comparison due to its enormous performance variability. We should highlight that the root causes of this high variability, which we already discussed (i.e., a thread may be randomly mapped to any core type, no effort is made to accurately track and balance the progress on AMPs, etc.), are still present in the stock Linux scheduler implementation from kernel versions that are more recent than the one we used (v3.10.104). Despite the various changes made to the Linux scheduler in newer kernels, none of these changes was made to address these AMP-related issues. Note that we also conducted the same set of experiments with CFS on Linux v4.16.1 –the latest stable version available at the time of this writing–, and observed the same huge performance variability.

For the `scp` benchmark, CAMPS and CFS yield similar slowdown figures in the F and D scenarios. We found that the (modest) variation in this case is not up to the OS scheduler itself (the benchmark is I/O intensive), but instead has to do with the disparities in the network bandwidth achieved by the different co-running instances of `scp`.

For the HPC workloads (last five groups of boxes in Fig. 6a) both schedulers provide very similar slowdown (in a 2 percent range), with only one exception: the `semphy` program. This program goes through parallel phases and long-term serial execution phases, wherein the application exposes a single runnable thread to the OS. During sequential phases, CAMPS maps the single runnable thread to a big core, and as a result, it effectively mitigates this scalability bottleneck [15], [34]. CFS only accelerates serial phases in the event that the thread responsible from running serial code happened to be already mapped to a big core (by chance).

We now turn our attention to the results of the HMP scheduler on 4B-4S (Fig. 6d). On this system, we could not gather the results for the `bzip` (D) and `kernbench` (D) workloads due to exceeding the platform's memory constraints (as discussed in Section 5.1.2). Unlike CAMPS, HMP is not designed to enforce system-wide fairness, but instead it extends CFS to provide a good tradeoff between performance and energy consumptions for mobile workloads on ARM big.LITTLE systems. To this end, it devotes big cores to run compute-intensive code, and uses small cores for interactive or I/O intensive applications. While CAMPS exhibits a similar (low variability) profile as that observed on 2B-2S, HMP does not behave exactly as CFS. In particular, in the H scenario, it nearly matches the slowdown distribution of CAMPS, as it uses big cores to run the various threads as soon as it detects they are going through a compute-intensive phase. In the F and D scenarios, HMP makes no effort to guarantee equal progress as opposed to CAMPS; HMP is subject to high variability, making it specially unsuitable to be considered as a baseline for comparison under long-running compute-intensive workloads.

For the I/O intensive workloads (`scp`), HMP delivers a smaller variance than our scheduling proposal (CAMPS was not optimized for I/O benchmarks), but it clearly delivers worse performance than CAMPS in the D scenario.

As for the multithreaded HPC programs (last five group of boxes), HMP yields very poor performance and is subject to large performance variability in some cases. Essentially, in these workloads HMP always maps all threads to big cores, thus introducing oversubscription (2 threads per big core) while leaving small cores idle. This mapping is very inappropriate in this context, as threads in some of these applications synchronize with each other frequently.

We now proceed to analyze the results of `ebizzy` on both platforms (Figs. 6b and 6e). Since CAMPS performs no worse than HMP and CFS in the H scenario, we focus on the discussion of the remaining cases. Note that threads of this web-server-like application do not synchronize with each other, but instead attempt to complete as many requests as possible in parallel. In the F and D scenarios, CFS effectively utilizes both core types. The fact that threads do not make equal progress does not affect throughput; big-core and small-core threads do effective work, but at a different pace. CAMPS, which is not optimized for this workload type, delivers a slightly inferior throughput (1.6 percent less) than CFS under F and D. Because `ebizzy` threads are CPU bound, HMP assigns them all to big cores, leading to poor performance. CAMPS is able to outperform HMP by 21 and 37 percent on average in the F and D scenarios.

Finally, we examine the tail scheduler latency numbers (Figs. 6c and 6f) obtained with multiple runs of `schbench`. To fully understand the results, it is worth recalling that CAMPS attempts to maximize big core utilization, and, to this end, it quickly moves threads to underloaded big cores. In the H scenario, CAMPS maps all threads to the available big cores, while CFS and HMP may leave some threads running on small cores. Big core's superior performance translates into smaller latencies, thus enabling CAMPS to outperform the other schedulers: it achieves a 65 and 45 percent latency reduction versus CFS and HMP, respectively. Our results also register more consistent tail latencies under CAMPS across different executions relative to CFS, but they also reveal a slightly worse tail latency (around 7 percent higher) under oversubscription (D). Lastly, we also observe that CAMPS's load balancing decisions in the F and D scenarios allow it to impressively reduce tail latencies (by up to 53 percent) w.r.t. HMP, which overloads big cores with compute-intensive threads.

6 Conclusions

In this paper, we have proposed CAMPS, an OS-level fairness-aware scheduler for asymmetric single-ISA multicores. Unlike other fairness-conscious asymmetry-aware schemes [11], [23], [24], our approach effectively caters to the performance degradation that comes from contention on the shared resources among cores, such as the last-level cache or the memory bus. CAMPS accurately tracks the progress that the various threads in the workload make when running on the different core types throughout the execution, and enforces fairness by evening out the progress across threads.

CAMPS's progress tracking scheme relies on approximating the current slowdown of an application thread by comparing its actual performance with the performance observed in the past for the thread when it ran on a big core in a low contention scenario. In doing so, the scheduler factors in the contention-related performance degradation as well as the slowdown that the thread normally experiences when it is mapped to a small core rather than to a big one. Notably, our approach does not require special hardware extensions [14], [23] or platform-specific speedup-prediction models [10], [11] to function. Instead, CAMPS relies on the gathering of a set of performance metrics that can be easily measured online in commercial AMP hardware via performance counters. This makes the scheduler highly portable across different processor models and CPU architectures.

We implemented CAMPS in the Linux kernel and assessed its effectiveness on real asymmetric hardware. An extensive comparison was performed with other existing schemes that aim to optimize fairness [11], [23], [24]. Our experimental results reveal that CAMPS outperforms the state-of-the-art fairness-aware scheme for AMPs –the ACFS scheduler [11]– in both fairness and throughput.

Acknowledgments

This work has been supported by the EU (FEDER) and the Spanish MINECO, under grant TIN 2015-65277-R. We would like to thank David Koufaty (Intel Labs) for enabling us to experiment with the QuickIA prototype system.

References

[1] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 45:1–45:38, Feb. 2016.
[2] T. Li, et al., "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *Proc. 16th Int. Symp. High-Performance Comput. Archit.*, 2010, pp. 1–12.
[3] J. Cong, et al., "Accelerator-rich architectures: Opportunities and progresses," in *Proc. 51st Ann. Design Autom. Conf.*, 2014, pp. 180:1–180:6.
[4] C. R. Johns and D. A. Brokenshire, "Introduction to the cell broadband engine architecture," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 503–519, Sep. 2007.
[5] J. P. Perez, et al., "CellSs: Making it easier to program the cell broadband engine processor," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 593–604, 2007.
[6] R. Kumar, et al., "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proc. 31st Ann. Int. Symp. Comput. Archit.*, 2004, pp. 64–75.
[7] ARM, "Benefits of the big.LITTLE Architecture." [Online]. Available: http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architect ure.pdf, Accessed on: Jan. 10, 2015.
[8] ARM, "Juno platform." [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.subset.boards.juno/index.html, Accessed on: Mar. 9, 2017.
[9] N. Chitlur, et al., "QuickIA: Exploring heterogeneous architectures on real prototypes," in *Proc. 39th Ann. Int. Symp. Comput. Arch.*, 2012, pp. 1–8.
[10] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. Eurosys*, 2010, pp. 125–138.
[11] J. C. Saez, et al., "Towards completely fair scheduling on asymmetric single-ISA multicore processors," *J. Parallel Distrib. Comput.*, vol. 102, pp. 115–131, 2017.
[12] D. Shelepov, et al., "HASS: A scheduler for heterogeneous multicore systems," *Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, 2009.
[13] J. C. Saez, et al., "Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems," *ACM Trans. Comput. Syst.*, vol. 30, no. 2, pp. 6:1–6:38, Apr. 2012.
[14] K. Van Craeynest, et al., "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proc. 39th Ann. Int. Symp. Comput. Archit.*, Jun. 9–13, 2012, pp. 213–224.
[15] J. C. Saez, et al., "Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors," in *Proc. 7th Int. Conf. Comput. Frontiers*, 2010, pp. 31–40.
[16] J. A. Joao, et al., "Utility-based acceleration of multithreaded applications on asymmetric CMPs," in *Proc. 40th Ann. Int. Symp. Comput. Archit.*, 2013, pp. 154–165.
[17] N. Markovic, et al., "Thread lock section-aware scheduling on asymmetric single-ISA multi-core," *IEEE Comput. Archit. Lett.*, vol. 14, no. 2, pp. 160–163, Jul. 2015.
[18] I. Jibaja, et al., "Portable performance on asymmetric multicore processors," in *Proc. Int. Symp. Code Generation Optimization*, 2016, pp. 24–35.
[19] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proc. 40th Ann. IEEE/ACM Int. Symp. Microarchitecture*, 2007, pp. 146–160.
[20] E. Ebrahimi, et al., "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *Proc. 15th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2010, pp. 335–346.
[21] H. Yun, et al., "Memory bandwidth management for efficient performance isolation in multi-core platforms," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 562–576, Feb. 2016.
[22] J. Feliu, et al., "Perf & fair: A progress-aware scheduler to enhance performance and fairness in SMT multicores," *IEEE Trans. Comput.*, vol. 66, no. 5, pp. 905–911, May 2017.
[23] K. Van Craeynest, et al., "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Tech.*, 2013, pp. 177–187.
[24] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proc. 3rd Int. Conf. Comput. Frontiers*, 2006, pp. 29–40.
[25] Hardkernel, "Odroid XU4 board," 2016. [Online]. Available: http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu4, Accessed on: Jun. 22, 2016.
[26] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 4, pp. 8:1–8:45, Dec. 2010.
[27] D. Xu, et al., "Providing fairness on shared-memory multiprocessors via process scheduling," in *Proc. ACM Int. Conf. Meas. Modeling Comput. Syst.*, 2012, pp. 295–306.
[28] S. Zhuravlev, et al., "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 4:1–4:28, Dec. 2012.
[29] H. Yun, et al., "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *Proc. 20th Real-Time Embedded Tech. Appl. Symp.*, 2014, pp. 155–166.
[30] Y. Ye, et al., "MARACAS: A real-time multicore vcpu scheduling framework," in *Proc. IEEE Real-Time Syst. Symp.*, 2016, pp. 179–190.
[31] A. Alhammad and R. Pellizzoni, "Trading cores for memory bandwidth in real-time systems," in *Proc. 22nd Real-Time Embedded Tech. Appl. Symp.*, Apr. 2016, pp. 1–11.
[32] M. Pricopi, et al., "Power-performance modeling on asymmetric multi-cores," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2013, pp. 15:1–15:10.
[33] M. Rasmussen, "Task placement for heterogeneous MP systems," 2012. [Online]. Available: https://lwn.net/Articles/517250/, Accessed on: Jul. 6, 2016.
[34] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *IEEE Comput.*, vol. 41, no. 7, pp. 33–38, Jul. 2008.
[35] V. Petrucci, et al., "Octopus-man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers," in *Proc. 21st Int. Symp. High-Performance Comp. Archit.*, 2015, pp. 246–258.
[36] M. E. Haque and others, "Exploiting heterogeneity for tail latency and energy efficiency," in *Proc. 50th Ann. IEEE/ACM Int. Symp. Microarchitecture*, 2017, pp. 625–638.

[37] Y. G. Kim, M. Kim, and S. W. Chung, "Enhancing energy efficiency of multimedia applications in heterogeneous mobile multi-core processors," *IEEE Trans. Comput.*, vol. 66, no. 11, pp. 1878–1889, Nov. 2017.

[38] J. C. Saez, et al., "On the interplay between throughput, fairness and energy efficiency on asymmetric multicore processors," *Comput. J.*, vol. 61, no. 1, pp. 74–94, 2018.

[39] C. Kim and J. Huh, "Fairness-oriented OS scheduling support for multicore systems," in *Proc. Int. Conf. Supercomputing*, 2016, pp. 29:1–29:12.

[40] X. Fan, Y. Sui, and J. Xue, "Contention-aware scheduling for asymmetric multicore processors," in *Proc. Int. Conf. Parallel Distrib. Syst.*, Dec. 2015, pp. 742–751.

[41] S. Barati and H. Hoffmann, "Providing fairness in heterogeneous multicores with a predictive, adaptive scheduler," in *Proc. Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 38–49.

[42] J. C. Saez, et al., "PMCTrack: Delivering performance monitoring counter support to the OS scheduler," *Comput. J.*, vol. 60, no. 1, pp. 60–85, 2017.

[43] S. Zhuravlev, et al., "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 4:1–4:28, Dec. 2012.

[44] M. Nabelsee, et al., "Load-aware scheduling for heterogeneous multi-core systems," in *Proc. 31st Ann. ACM Symp. Appl. Comput.*, 2016, pp. 1844–1851.

[45] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," in *19th Int. Conf. Parallel Archit. Compilation Tech.*, 2010, pp. 237–248.

[46] A. Annamalai, et al., "An opportunistic prediction-based thread scheduling to maximize throughput/watt in AMPs," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Tech.*, 2013, pp. 63–72.

[47] F. Cerqueira, et al., "Linux's processor affinity API, refined: Shifting real-time tasks towards higher schedulability," in *Proc. IEEE Real-Time Syst. Symp.*, 2014, pp. 249–259.

[48] T. Li, A. R. Lebeck, and D. J. Sorin, "Spin detection n hardware for improved management of multithreaded systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 6, pp. 508–521, Jun. 2006.

[49] Linux test project. [Online]. Available: https://github.com/linux-test-project/ltp

[50] M. Fleming, "A survey of scheduler benchmarks," 2017. [Online]. Available: https://lwn.net/Articles/725238/, Accessed on: Jan. 20, 2018.

Adrian Garcia-Garcia received the MSc degree in computer science from the Complutense University of Madrid (UCM), in 2018. He is now working toward the PhD degree at UCM. His current research interests include OS scheduling, and on the analysis of shared resource contention effects on multicore systems. His work is supported by a UCM research fellowship grant.

Juan Carlos Saez received the PhD degree in computer science from the Complutense University of Madrid (UCM), in 2011. He is now an associate professor with the Department of Computer Architecture, UCM. His research interests include energy-aware computing and improving the interaction between the OS and hardware for emerging architectures. His recent research interests include OS scheduling on asymmetric multicores, exploring new techniques to deliver better performance per watt, and QoS on these systems.

Manuel Prieto-Matias received the PhD degree from the Complutense University of Madrid (UCM), in 2000. He is now associate professor with the Department of Computer Architecture, UCM. His research interests include parallel computing and computer architecture. His current research addresses emerging issues related to asymmetric processors, heterogeneous systems and energy-aware computing, with a special emphasis on the interaction between the OS and the underlying architecture. He has co-written numerous articles in journals and international conferences in the field of parallel computing and computer architecture.