




Delivering Fairness on Asymmetric Multicore Systems via Contention-Aware Scheduling

Adrian Garcia-Garcia , Juan Carlos Saez ^(✉) , and Manuel Prieto-Matias 

Facultad de Informática, Complutense University of Madrid, Madrid, Spain
{adriagar,jcsaezal,mpmatias}@ucm.es

Abstract. Asymmetric single-ISA multicore processors (AMPs), which integrate high-performance big cores and low-power small cores, were shown to deliver better energy efficiency than symmetric multicores for diverse workloads. Previous work has highlighted that this potential of AMP systems can be realizable with help from the OS scheduler. Notably, delivering fairness on AMPs still constitutes an important challenge, as it requires the scheduler to accurately track the progress of each thread as it runs on the various core types throughout the execution. In turn, this progress depends on the speedup that an application derives on a big core relative to a small one. 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. Our experimental evaluation, which employs real asymmetric hardware and scheduler implementations in the Linux kernel, demonstrates that CAMPS improves fairness by 10.6% on average with respect to a state-of-the-art fairness-aware scheme, while delivering higher throughput.

Keywords: Asymmetric multicore · OS scheduling · Fairness
Linux kernel

1 Introduction

Previous research has shown that asymmetric single-ISA (instruction set architecture) multicore processors (AMPs), which integrate a mix of complex high-performance big cores and power-efficient small cores on the same chip, can deliver higher performance per watt than their symmetric counterparts for diverse workloads [8, 15]. To bring the potential of AMPs to unmodified applications, the operating system has to face a number of challenges [9], some of which must be properly addressed by the OS scheduler [10].

Most asymmetry-aware schedulers have been designed to optimize the system throughput for multi-application workloads [3, 7, 8, 12]. To this end, the scheduler must devote big cores to running applications that use these cores efficiently,

since they derive performance improvements (speedup) relative to running on small cores [8]. Further throughput gains can be obtained by using big cores to accelerate different scalability bottlenecks present in parallel programs [6, 12].

Unfortunately, asymmetry-aware schedulers that strive to optimize throughput alone are known to be inherently unfair [14]. Unfairness gives rise to a number of undesirable effects on multicore systems [5, 18]. For example, equal-priority applications may not experience the same performance degradation (slowdown) when running together relative to the performance observed when each application runs alone on the AMP. Moreover, when attempting to optimize throughput, the completion time of an application on an AMP may largely depend on its co-runners [14]. These issues make priority-based scheduling policies ineffective, reduce performance predictability, and can lead to wrong billings in commercial cloud-like computing services, where users are charged for CPU hours.

These QoS-related issues can be addressed on AMPs via fairness-aware scheduling algorithms [3, 9, 14, 16]. Most of these algorithms rely on tracking the progress that individual threads make when running on the various core types throughout the execution, and attempt to deliver fairness by swapping threads between different cores based on the observed progress. In tracking progress, existing schedulers [14, 16] factor in the slowdown that a thread experiences when it is mapped to a small core, which can differ greatly across applications and vary over time as a program goes through different execution phases [3, 12]. Notably, these schedulers do not consider the performance degradation that comes from contention on the shared resources among cores, which may also lead to unfairness [5, 20]. In current AMP hardware [2, 4], clusters of cores of the same type typically share a last-level cache and other memory-related resources. Applications running on the various cores may contend for shared resources, which could degrade their performance in an uneven and unpredictable way [5, 18–20].

To address this shortcoming, we propose CAMPS, an OS-level contention-aware scheduler for AMPs that seeks to optimize fairness while maintaining acceptable throughput. CAMPS is equipped with a novel mechanism to approximate a thread’s current slowdown, which leverages past performance history gathered at runtime in low contention scenarios. Unlike other schedulers, CAMPS does not need special hardware extensions [16] or platform-specific prediction models [7, 12, 14] to function. Instead, it relies on performance counters available in commercial hardware, which makes the scheduler highly portable across architectures. To assess the effectiveness of our proposal, we implemented it in the Linux kernel and evaluated it on a real AMP platform that features an ARM big.LITTLE processor [2]. Our analysis reveals that CAMPS improves fairness by 10.6% on average compared to a state-of-the-art fairness-aware scheduling scheme [14], and at the same time improves throughput by up to 17%.

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

2 Background and Related Work

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

2.1 Fairness on AMPs and Determining the Slowdown

Previous research on fairness for CMPs [5, 18] and AMPs [6, 14, 16] define a scheme as fair if equal-priority applications in a multi-program workload suffer the same slowdown due to sharing the system. To cope with this notion of fairness, we turned to the lower-is-better *unfairness* metric [5]:

$$Unfairness = \frac{MAX(Slowdown_1, \dots, Slowdown_n)}{MIN(Slowdown_1, \dots, Slowdown_n)} \quad (1)$$

where n is the number of applications in the workload and $Slowdown_i = CT_{sched,i}/CT_{alone,i}$. 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:

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

where IPS_{alone} represents 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 the context of a multi-program workload under a given scheduling algorithm.

In this work, we assume that the IPS_{alone} on an AMP is maximized when the thread runs on a high-performance big core in isolation. That is the case across all the applications explored in our experiments. We should also highlight that in the context of multi-threaded programs, the IPS can be a somewhat misleading performance metric, since a thread can exhibit a high IPC when busy waiting (spinning) for other threads to arrive at a synchronization point (e.g. barrier). To make the OS scheduler aware of these situations, where threads do no useful work, our scheduling scheme leverages spin notifications from the user-level runtime system by following a similar approach to that proposed in [13].

Delivering fairness entails ensuring that the slowdown accumulated by the various application threads throughout the execution remains as even as possible [5, 14, 16, 18], while maintaining acceptable throughput. To this end, the scheduler must be equipped with a mechanism to determine a thread's slowdown at runtime. However, measuring the slowdown directly by using Eq. 2 is difficult in practice; while a thread's IPS_{sched} can be easily obtained via performance

counters, accurately determining IPS_{alone} online is a challenging task, even on symmetric CMPs [20]. For that reason, existing scheduling algorithms for symmetric CMPs typically rely on estimation models to approximate IPS_{alone} [18], or employ different heuristics to determine the degree of performance degradation indirectly via contention-related metrics [20]. Unfortunately, these scheduling algorithms are not suitable for AMPs, as they assume that the 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 the system. On current AMP hardware [2, 4, 15], this assumption is not valid, as cores may exhibit different microarchitectural features and cache sizes [7, 14].

Recently proposed fairness-aware schedulers for AMPs [14, 16], implicitly rely on the assumption that the performance degradation experienced by a thread on an AMP (relative to its solo execution) is negligible when it runs on a big core, even if it runs simultaneously with other threads. Thus, a thread’s slowdown is estimated to be 1 when it runs on a big core; and the thread’s big-to-small performance ratio – also referred to as the *speedup factor* (SF) [12] – is used to approximate the slowdown when the thread runs on a small core. In turn, the SF can be determined online by various means, such as direct measurement (IPC sampling) [3, 8], prediction models based on hardware counters [7, 12, 14] or by leveraging special hardware extensions [16].

2.2 Performance Impact of Shared Resource Contention on AMPs

Assuming that a thread’s slowdown is negligible when it runs on a big core (as done in [14, 16]) is unrealistic in scenarios where threads heavily contend for shared resources with each other. To illustrate this fact, we analyzed the slowdown experienced by different single-threaded applications under varying degree of contention. Our analysis reveals that contention-related degradation can be substantial, and should be accounted for to avoid unfairness.

For our experiment, we used two AMP configurations based on the ARM Juno development board [2] – equipped with a mix of Cortex A57 and Cortex A53 cores, and the Intel QuickIA prototype [4], a dual-socket system featuring an Intel Atom N330 processor and a Xeon E5450 processor. The ARM-based configuration – presented in more detail in Sect. 4, features two big cores and four little cores. The Intel-based configuration integrates two big and two small cores. On both asymmetric platforms, 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 (FSB on Intel, AMBA on ARM) with the remaining cores in the cluster. Both platforms integrate a single DRAM controller shared among all cores.

Our experiment consists in measuring the slowdown experienced by diverse programs when mapped to a big core and run simultaneously with a different number of instances of an aggressor application. As the aggressor, we used the **bandwidth** benchmark [19], which causes substantial contention on the LLCs, shared buses and DRAM controller. On our platforms, we observed that this benchmark is capable of causing even a higher degree of contention than the one generated by highly memory-intensive SPEC CPU benchmarks, such as **1bm**.

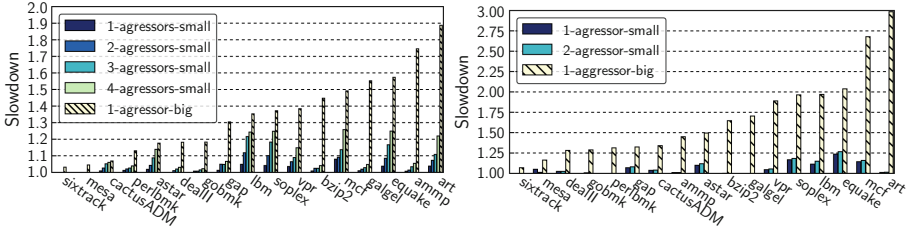


Fig. 1. Slowdown experienced by various benchmarks when running together with several instances of `bandwidth` on the Juno board (left) and the Intel QuickIA (right).

Figure 1 shows the slowdown (relative to the solo execution) that different applications experience when running simultaneously with several instances of the `bandwidth` application. Note that we measured the slowdown for all benchmarks in the SPEC CPU2000 and CPU2006 suites, but due to space constraints we only display the results for a few representative benchmarks that cover the full spectrum of slowdown values observed on both AMP platforms. 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” in Fig. 1, the benchmark runs simultaneously with one instance of `bandwidth`, which is also mapped to a big core; the small cores remain idle in this case. In the remaining scenarios, labeled as “ N -aggressors-small”, N instances of `bandwidth` are mapped to small cores; thus, in leaving one big core unused, we remove contention on the LLC and the bus interface associated with the big core cluster, but not on the DRAM controller (shared among all cores).

As is evident, the performance penalty that a thread may suffer on a big core due to interference with memory-intensive threads mapped to big cores is much greater (up to 1.89x on the ARM platform, and 2.98x on the Intel platform) than the degradation that comes from placing multiple aggressors on small cores (up to 1.26x, reached with the highest number of simultaneous small-core aggressors possible). This stems from two main factors. 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 memory resources is higher when it runs on a big core than on a small one. We hypothesized that this has to do with the fact that in-order small cores cannot handle multiple outstanding cache misses, leading to a smaller bus and memory bandwidth utilization, and as a result to a smaller degree of contention. This observation suggests that monitoring the IPS of a thread when it runs on a big core in a contention-free scenario on a big cluster (e.g. with the other big cores idle) could be a good estimate for IPS_{alone} . Our scheduling proposal leverages this observation to approximate the slowdown.

We also observe that some programs, such as `sixtrack` or `mesa`, experience very low slowdown when executed together with memory-intensive aggressors. As pointed out in [18,20], CPU-intensive applications with a very small

working set 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 [18], 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. In our platforms, the BTR is measured as follows: $(bus_read_accesses * LLC_cache_line_size * processor_freq) / total_cycle_count$.

2.3 Related Work

The first approach to fairness-aware scheduling on AMPs was an asymmetry-aware Round-Robin (RR) scheduler that simply fair-shares big cores among applications by triggering periodic thread migrations [3]. Fair-sharing big cores has proven to provide better performance and more repeatable completion times across runs on AMPs than default schedulers in general-purpose OSe [9, 11], which are largely asymmetry agnostic. For this reason, RR has been widely used as a baseline for comparison [3, 12]. Note, however, that RR and other schemes that also rely on fair sharing big cores, such as A-DWRR [9] do not take into account the fact that applications derive different speedup factors when using big cores on the platform, and that these speedups may vary over time. This leads to degrading fairness and throughput [14].

Currently, the state-of-the-art OS-level fairness-aware scheduling scheme is ACFS [14]. To optimize fairness, ACFS leverages per-thread speedup factor (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. A thread's SF is determined online by feeding a platform-specific estimation model with the values of different performance metrics gathered via hardware counters. In [14] the authors experimentally demonstrated that ACFS clearly outperforms previous fairness-aware scheduling schemes, such as RR [3], Equal-Progress [16], and A-DWRR [9], for a wide range of workloads running on real asymmetric hardware. The main limitation of ACFS [14] (also present in previous schemes [16] based on thread progress tracking mechanisms), is the fact that the scheduler does not take shared-resource contention effects into consideration. As our experiments reveal, failing to cater to the degradation that comes from contention leads the scheduler to exhibit unfair behavior when multiple memory-intensive programs are included in the workload. CAMPS effectively improves fairness in this scenario.

3 The CAMPS Scheduler

CAMPS consists of two components: the *performance monitor* and the *core scheduler*. The performance monitor continuously 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 in the system, and swaps threads between cores when necessary to ensure that applications achieve similar progress on the AMP.

In the remainder of this section we first describe the mechanism used by CAMPS to predict a thread’s slowdown at runtime. Then we outline the progress tracking mechanism and discuss how fairness is enforced via thread swaps.

3.1 Determining the Slowdown at Runtime

The *performance monitor* approximates a thread’s current slowdown by using Eq. 2; the actual IPS is measured with performance counters, and 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 was mapped to a big core in a low-contention scenario. As shown in Sect. 2, when a thread runs on a big core, the performance degradation that comes from interference with small-core threads is typically very low. Based on this observation, big-core low-contention IPS values are used to approximate the IPS_{alone} .

To detect low-contention scenarios on a big core, the scheduler leverages the heuristics based on the bus transfer rate (BTR) metric proposed in [17, 18]. Essentially, a thread whose BTR is smaller than a given *low_btr* threshold are not likely to suffer noticeably from contention. In a similar vein, when the aggregate BTR across threads running a given core cluster falls below a given *high_btr* threshold we can assume that degradation due to contention will be very low [18]. These thresholds can be quickly determined via synthetic benchmarks [17, 18]. If low-contention scenarios do not occur naturally as a result of the thread-to-core assignments performed by CAMPS, the core scheduler will enter a special mode (described later), 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 program phase already exists in the table or not. To this end, we leverage a variant of the phase-detection mechanism used in a previous work [1]. In that work, the scheduler continuously monitors the percentage of instructions of different types (int/FP, load, store, branches, etc.) retired during the last sampling period, which make up a *instruction type vector* (ITV). Specifically, if the Manhattan distance of the ITVs for two performance samples (collected at different intervals) is smaller than a threshold, then both samples are assumed to belong to the same phase. Unfortunately, this scheme cannot be implemented in the real AMP platform we used, as the Performance Monitoring Unit is not equipped with the necessary performance events. To overcome this issue, we adapted this approach by monitoring two alternative *control metrics* along with the thread’s BTR and its IPS: the number of L1 cache accesses per 1K instructions, and the percentage of branches retired over the total instruction count. As the instruction composition, the value of these two control metrics for a specific phase remain the same under different levels of shared resource contention, and more importantly, they do not vary significantly across core types. In addition, we observed that the value of these metrics changes dramatically when an application enters a new execution phase exhibiting a different degree of memory intensity and branch-prediction related behavior, which have a great impact on cross-core performance on AMPs [7, 14]. These facts make the selected *control metrics* very suitable to index the table.

The history 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 there is not any information of the current phase, a new IPS entry is created; otherwise the existing is updated with a running average of the low-contention IPS values recorded for that phase. When the thread runs on a small core, or 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, and the current IPS value measured in the last sampling interval. Otherwise (i.e. *phase miss*), the slowdown is approximated with the ratio of the average IPS samples stored in the history table, and the current IPS value.

3.2 Progress Tracking and Enforcing Fairness

CAMPS's core scheduler maintains a progress counter for each thread referred to as `amp_progress`, which enables the scheduler to track progress and enforce fairness. 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 isolation. When a thread runs for a clock *tick* on a given core type, the scheduler increments `amp_progress` by $\Delta_{\text{amp_progress}}$, defined as follows:

$$\Delta_{\text{amp_progress}} = (100 \cdot W_{\text{def}}) / (CS \cdot W_t) \quad (3)$$

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*.

The definition of $\Delta_{\text{amp_progress}}$ is very similar to the formula that the ACFS scheduler [14] uses to update progress counters. The main difference lies in how the CS factor is defined. ACFS assumes that a thread's current slowdown is always 1 (no performance degradation) when it runs on a big core, and uses the thread speedup factor (predicted via a platform-specific model) to approximate the slowdown when the thread runs on a small core. In doing so, ACFS does not take shared resource contention into consideration when updating progress. This aspect is factored in by CAMPS, as the slowdown is determined by comparing the thread's actual performance with an estimate of IPS_{alone} .

Threads mapped to big cores by the scheduler typically make faster progress than threads running on small ones, which causes unfairness. Note that the CS factor (slowdown) is usually bigger when the thread runs on a small core; thus, progress counters of small-core threads are incremented at a slower pace than that of big-core threads. CAMPS strives to enforce fairness by evening out the progress counter across threads. To make this possible, it may need to perform thread swaps (migrations) between different core types every so often. Like ACFS, CAMPS swaps a thread running on a big core with another thread running on a small core when the difference of their progress counters exceeds a given threshold. Specific instructions are provided in [14] for selecting the most appropriate value of this threshold for a given platform.

We found that relying on progress counters alone (as ACFS does) is ineffective in the event that a contention-sensitive application and an aggressor are mapped to the big-core cluster simultaneously. As shown in Sect. 2, this may slow down contention-sensitive applications substantially. To overcome this issue, CAMPS uses the BTR-based heuristics [18] to detect high contention scenarios, and favors those threads swaps that contribute to reducing contention on the big core cluster (e.g. a big-core aggressor thread is migrated to a small core). The main goal of this is to reduce the slowdown experienced by threads mapped to the big-core cluster simultaneously, and in turn, to improve fairness and throughput.

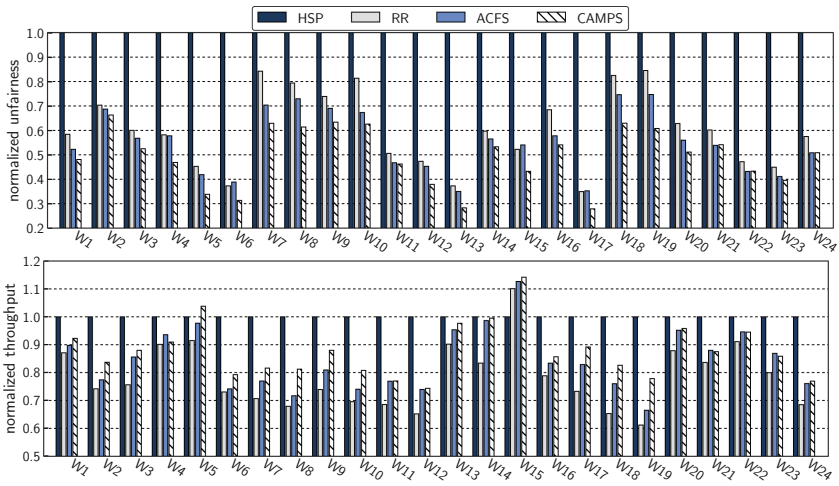
Finally, it is worth noting that when the number of memory-intensive threads in the workload is high, low contention scenarios may not occur that often. In these cases, CAMPS transitions into a non-work-conserving (NWC) mode in which low contention scenarios are created artificially. To control transitions into this special mode, the scheduler operates as follows. Every time that a thread completes k consecutive monitoring intervals (being k a configurable parameter), CAMPS calculates the thread’s phase-hit rate and the number of IPS samples that were inserted into the history table over that time period. If the phase-hit rate is not high enough (falls below 80% in our experimental platform) and not a single IPS sample was inserted in the table during that period, the scheduler enters the NWC mode. When in this mode, if the thread was not running on a big core already, it will be swapped with a big-core thread to preserve load balance; in doing so, CAMPS tries to select a memory-intensive thread as the swap partner, so as to reduce contention on the big core cluster. If a low-contention scenario is still not present on the big-core cluster, the scheduler will disable as many big cores as necessary (for a very short period of time) to mimic such a scenario. Making this possible comes down to disabling only a few big cores: those where potentially aggressor (high-BTR) threads are running at this point. The scheduler transitions back into the normal operating mode when (1) a number of IPS samples have been gathered, or (2) when the thread blocks/exits.

4 Experimental Evaluation

We compare the effectiveness of CAMPS with that of two previously proposed fairness-aware schedulers for AMPs: ACFS [14] and an asymmetry-aware Round-Robin (RR) scheme [3]. We opted to use RR instead of the default OS scheduler, which is known to deliver highly variable completion times for compute-intensive workloads [11]. For the sake of completeness, 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 [7, 12]. We will refer to this scheduler as *HSP* (High SPeedup). All the schemes considered in our study were implemented as a separate scheduling class in the Linux kernel v3.10. Except for RR, all the schedulers rely on performance monitoring counters (PMCs) to function. Our implementation of HSP and ACFS determine threads’ speedup factors on-line by monitoring different PMC events, and by feeding an estimation model with the obtained event counts, as described in [11].

Table 1. Multi-application workloads

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

**Fig. 2.** Unfairness and throughput for workloads in Table 1

To assess the effectiveness of the various algorithms, we employed multi-application workloads consisting of compute-intensive benchmarks from different benchmarks suites (SPEC CPU, PARSEC, etc.) running on two real AMP platforms with different number of cores. Due to space constraints however, we could only include the discussion for the results of workloads consisting of single-threaded programs, and running on the ARM Juno board [2]. In using this kind of workloads, we ensure a fair comparison against HSP, RR and ACFS, as these schemes were evaluated before using similar workloads [3, 7, 14]. The ARM Juno board used for our experiments features a big.LITTLE processor that consists of two Cortex A57 “big” cores (running at 1.10 GHz) and four Cortex A53 “small” cores (running at 850 MHz). Each core has a private L1 cache and shares a last-level (L2) cache with the other cores of the same type. Specifically, big cores share a 2 MB/16-way L2 cache, and small ones feature a 1 MB/16-way cache.

For the evaluation on the Juno board, we randomly built 24 program mixes that combine a different number of *light-sharing* programs – whose performance does not suffer noticeably under contention, and *memory-intensive* programs, which are subject to high contention-related performance degradation or put significant pressure on shared resources. Table 1 displays the workloads sorted in descending order by the number of memory-intensive programs they include. Figure 2 reports the unfairness and throughput values for each workload and scheduler, normalized with respect to the results of the HSP scheduler. To assess throughput we employed the *Aggregate Speedup* (ASP) metric as in [11, 14].

The results illustrate that optimizing one metric may lead to substantial degradation of the other metric. This is consistent with what was observed in previous work [11, 14], which underscores that fairness and throughput are largely conflicting optimization goals on AMPs. As is evident, the HSP scheduler, which strives to optimize throughput achieves the best ASP values for the 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 vs. HSP (up to 72% – CAMPS under W17), at the cost of potentially high throughput degradation (up to 38% – RR, W19).

ACFS, RR and CAMPS exhibit a clear trend across the board. Specifically, for most 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 take contention effects into consideration, leads it similar unfairness figures to those of RR in some cases (e.g. W4-W6, W15 or W17). By contrast, our proposal (CAMPS) is able to reduce unfairness even further: by up to 11% with respect to ACFS (W17) and by up to 28% relative to RR (W19). At the same time, CAMPS is capable to reap higher throughput gains. Notably, under those workloads with a low degree of contention (W20-W24) – due to the small number of memory-intensive applications, CAMPS and ACFS perform very similarly. This demonstrates that our proposal is also suitable for low-contention scenarios, as it delivers similar unfairness and throughput figures to ACFS, which provides the best results under these circumstances [14]. All in all, CAMPS achieves an average 10.6% reduction in unfairness with respect to ACFS while improving throughput by up to 17% (4% average increase).

Finally, we also observed that HSP is especially affected by contention effects under workloads W5 and W13-W15, where the two applications with the highest speedup constitute a pair consisting of an aggressor and a contention-sensitive program. In these cases, HSP maps these conflicting applications simultaneously to the two available big cores very often. Despite the fact that the applications derive benefits from running on a big core alone, they also contend for shared resources, which gives rise to throughput degradation. Fairness-aware schedulers mitigate this issue by swapping threads between core types every so often, which reduces the amount of time that the conflicting applications are mapped together on the same cluster; this contributes to improving throughput. Specifically, the

results reveal that all fairness-aware schedulers reap high normalized throughput figures under these workloads (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 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 [3, 14, 16], our approach effectively caters to the performance degradation that comes from contention on shared resources among cores. 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. To this end, CAMPS approximates 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 low contention scenarios. Notably, our proposal does not require special hardware extensions [16] or platform-specific speedup-prediction models [7, 14] to function. Instead, CAMPS relies on performance counters available in commercial AMP platforms, which makes it portable across CPU architectures. We implemented CAMPS in the Linux kernel and assessed its effectiveness on a real AMP system that features an ARM big.LITTLE processor. An extensive comparison was performed with existing asymmetry-aware schedulers [3, 7, 14, 16]. Our experiments reveal that CAMPS outperforms the state-of-the-art fairness-aware scheme – ACFS [14] – in both fairness and throughput.

Acknowledgements. This work has been supported by the EU (FEDER) and the Spanish MINECO under grant TIN 2015-65277-R.

References

1. Annamalai, A., et al.: An opportunistic prediction-based thread scheduling to maximize throughput/watt in AMPS. In: Proceedings of PACT 2013, pp. 63–72 (2013)
2. ARM: Juno ARM development platform. <http://infocenter.arm.com/help/topic/com.arm.doc.subset.boards.juno/index.html> (2014)
3. Becchi, M., Crowley, P.: Dynamic thread assignment on heterogeneous multiprocessor architectures. In: Proceedings of CF 2006, pp. 29–40 (2006)
4. Chitlur, N., et al.: QuickIA: exploring heterogeneous architectures on real prototypes. In: Proceedings of HPCA 2012, pp. 1–8 (2012)
5. Ebrahimi, E., et al.: Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In: Proceedings of ASPLOS 2010, pp. 335–346 (2010)
6. Joao, J.A., et al.: Utility-based acceleration of multithreaded applications on asymmetric CMPs. In: Proceedings of ISCA, vol. 13, pp. 154–165 (2013)

7. Koufaty, D., et al.: Bias scheduling in heterogeneous multi-core architectures. In: Proceedings of EuroSys 2010, pp. 125–138 (2010)
8. Kumar, R., et al.: Single-ISA heterogeneous multi-core architectures for multi-threaded workload performance. In: Proceedings of ISCA 2004, pp. 64–75 (2004)
9. Li, T., et al.: Operating system support for overlapping-ISA heterogeneous multi-core architectures. In: Proceedings of HPCA 2010, pp. 1–12 (2010)
10. Mittal, S.: A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv.* **48**(3), 45:1–45:38 (2016)
11. Saez, J.C., et al.: On the interplay between throughput, fairness and energy efficiency on asymmetric multicore processors. *Comput. J.* (to appear)
12. Saez, J.C., et al.: A comprehensive scheduler for asymmetric multicore systems. In: Proceedings of EuroSys 2010, pp. 139–152 (2010)
13. Saez, J.C., et al.: Operating system support for mitigating software scalability bottlenecks on AMPs. In: Proceedings of the CF 2010, pp. 31–40 (2010)
14. Saez, J.C., et al.: Towards completely fair scheduling on asymmetric single-ISA multicore processors. *J. Parallel Distrib. Comput.* **102**, 115–131 (2017)
15. Samsung: benefits of the big.LITTLE architecture. <http://www.samsung.com/semiconductor/minisite/Exynos/data/benefits.pdf>. Accessed 10 Jan 2015
16. Van Craeynest, K., et al.: Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In: Proceedings of PACT 2013, pp. 177–187 (2013)
17. Xu, D., et al.: On mitigating memory bandwidth contention through bandwidth-aware scheduling. In: Proceedings of PACT 2010, pp. 237–248 (2010)
18. Xu, D., et al.: Providing fairness on shared-memory multiprocessors via process scheduling. In: Proceedings of SIGMETRICS 2012, pp. 295–306 (2012)
19. Yun, H., et al.: PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In: Proceedings of RTAS 2014, pp. 155–166. IEEE (2014)
20. Zhuravlev, S., et al.: Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.* **45**(1), 4:1–4:28 (2012)