J. Parallel Distrib. Comput. 102 (2017) 115–131

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Towards completely fair scheduling on asymmetric single-ISA multicore processors

Juan Carlos Saez [a,*], Adrian Pousa [b], Fernando Castro [a], Daniel Chaver [a], Manuel Prieto-Matias [a]

[a] Complutense University of Madrid, Facultad de Informática, Ciudad Universitaria s/n, Madrid 28040, Spain

[b] III-LIDI, Facultad de Informática, UNLP, Calles 50 y 120 - La Plata - Bs. As., Argentina

HIGHLIGHTS

- Throughput and fairness are largely conflicting optimization goals on AMPs.
- Previous asymmetry-aware schedulers fail to effectively deal with user priorities.
- ACFS achieves an average 23% fairness improvement over state-of-the-art schemes.
- Predicting cross-core performance on current AMPs is one of the major challenges.

ARTICLE INFO

Article history:
Received 21 July 2015
Received in revised form
5 November 2016
Accepted 5 December 2016
Available online 19 December 2016

Keywords:
Asymmetric multicore
Scheduling
Operating systems
Fairness
CFS
Linux kernel

ABSTRACT

Single-ISA asymmetric multicore processors (AMPs), which combine high-performance big cores with low-power small cores, were shown to deliver higher performance per watt than symmetric CMPs (Chip Multi-Processors). Previous work has highlighted that this potential of AMP systems can be realizable via OS scheduling. To date, most existing scheduling schemes for AMPs have been designed to optimize the system throughput, but they are inherently unfair. Although fairness-aware schedulers have been also proposed, they fail to effectively deal with user priorities and do not always ensure that equal-priority applications experience a similar slowdown. To overcome these limitations, we propose ACFS, an asymmetry-aware completely fair scheduler that seeks to optimize fairness while ensuring acceptable throughput. Our evaluation on real AMP hardware and using scheduler implementations in the Linux kernel demonstrates that ACFS achieves an average 23% fairness improvement over two state-of-the-art schemes, while providing higher system throughput.

1. Introduction

Single-ISA asymmetric CMPs (AMPs) combine several core types with the same instruction-set architecture but different microarchitectural features. Asymmetric designs have been shown to significantly improve energy and power efficiency over symmetric CMPs [15]. Notably, combining just two core types (high-performance *big* cores with low-power *small* ones) simplifies the design and is enough to obtain most benefits from AMPs [15]. The ARM big.LITTLE processor [2] and the Intel QuickIA prototype [4] demonstrate that this design approach has drawn the attention of major hardware players.

Despite the potential benefits of AMPs, effectively utilizing the various cores types on the platform constitutes a significant challenge to the system software. This task can be accomplished by the OS scheduler [31,13,27]. Most existing scheduling schemes have focused on maximizing the system throughput [15,31,13,27]. To this end, the scheduler needs to map to big cores those applications that derive a high performance improvement from running on these cores relative to small ones [15]. Further throughput gains can be achieved by using big cores to accelerate sequential phases of parallel programs [11,27].

Maximizing throughput alone, however, may give rise to various issues. First, an application may experience very different completion time from run to run, since in one run it may be mapped to a big core the whole time and relegated to a small one in another depending on the co-running applications [17,31]. Second, the end

* Corresponding author.
E-mail addresses: jcsaezal@ucm.es (J.C. Saez), apousa@lidi.info.unlp.edu.ar (A. Pousa), fcastror@ucm.es (F. Castro), dani02@ucm.es (D. Chaver), mpmatias@ucm.es (M. Prieto-Matias).

http://dx.doi.org/10.1016/j.jpdc.2016.12.011
0743-7315/

user naturally expects that applications with equal priorities get equal slowdowns as a result of sharing the platform [6]. This is not usually the observed behavior on AMPs if the scheduler only seeks to maximize throughput [26]. Third, in scenarios with imposed QoS constraints trading throughput for fairness may be in order to improve user experience.

Previous research has shown that some of these issues can be mitigated via fairness-aware scheduling for AMPs [3,17,26,5]. In this work, we demonstrate that existing fairness schemes are either subject to high throughput degradation or do not constitute effective priority-based schemes. Notably, some of these techniques [3,17] do not exploit the fact that applications in a multi-program workload may derive a different benefit from using the big cores in the AMP. As a result, they fail to optimize fairness [29].

To address these shortcomings, we propose an *Asymmetry-aware Completely Fair Scheduler* (ACFS), which seeks to optimize fairness while maintaining acceptable system throughput. We implemented ACFS in the Linux kernel (on top of the asymmetry-agnostic Completely Fair Scheduler) and evaluated it using real asymmetric hardware. Our proposal effectively enforces user priorities and does not require hardware support or changes in the applications. A high-level description and reduced experimental analysis of the ACFS scheduler was presented in [30]. We extend that work with the following contributions:

- We carry out a theoretical study that illustrates the interrelationship between fairness and throughput on AMPs (Section 2.1). The simulator we built for that study, which relies on the theoretical model proposed in [29], was crucial for us to guide the design process of ACFS.
- We augmented the description of ACFS by showcasing key details necessary to reproduce our implementation of ACFS in a real OS, and by including clarifying examples on the internal workings of the scheduler.
- We propose a systematic methodology to build accurate estimation models enabling to approximate the relative performance benefit that a thread experiences on a big core relative to a small one (aka. speedup factor). ACFS and other schemes [26,13] rely on these models (based on hardware counters) to drive scheduling decisions. Existing methodologies to build these estimation models either require embedding off-line collected information in the application binary [32,24] or, as we demonstrate in this work, turn out ineffective for highly asymmetric systems [27], where cores exhibit profound microarchitectural differences, such as on the Intel QuickIA platform [4]. Our proposal overcomes these limitations. Moreover, to the best of our knowledge, our work is the first in deriving an accurate model to predict the speedup factor on the Intel QuickIA.
- We performed a substantial extension of the experimental analysis by including key sensitivity studies (as the ones discussed in Sections 3.3 and 5.4) and by experimenting with a much wider set of workloads under different state-of-the-art schedulers considered for the evaluation [13,27,17,5]. Note that some of these previously-proposed schemes were evaluated before using emulated asymmetric hardware [17] or simulators [5]. Instead, we performed an extensive evaluation on real asymmetric hardware, which enabled us to detect important drawbacks of the various schemes. Our evaluation reveals that ACFS reduces unfairness by 23% on average compared to other fairness-aware schedulers [17,5] and, at the same time, provides better system throughput than these schemes.

The rest of the paper is organized as follows. Section 2 discusses background and related work. Section 3 outlines the design of the ACFS scheduler. Section 4 presents our methodology to build speedup factor estimation models. Section 5 showcases our experimental results and Section 6 concludes.

Table 1
Synthetic workloads.

Workload	SF_1	SF_2	SF_3	SF_4
W1	4.7	4.7	1	1
W2	4.7	4.7	2.9	2.9
W3	4.3	4	4	1
W4	2.9	2.9	2.1	2.1
W5	4.7	4.7	3.6	3.6
W6	4.7	4.7	4.3	4.3
W7	3.2	2.5	1.7	1
W8	4	2.5	2.5	2.5
W9	4.7	4	3.2	2.5
W10	4	3.2	2.5	1

2. Background and related work

In this section we first analyze the interrelationship between fairness and system throughput on AMPs and then proceed to discuss related work.

2.1. Fairness and throughput on AMPs

Previous research on fairness for CMPs [8,21,6] and AMPs [26,5] 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 [6]:

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

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

To quantify throughput on AMPs, previous work [29] has employed the *Aggregate SPeedup* (ASP) metric, defined as follows:

$$ASP = \sum_{i=1}^{n} \left(\frac{CT_{slow,i}}{CT_{sched,i}} - 1 \right) \quad (2)$$

where $CT_{slow,i}$ is the completion time of application i when it runs alone on the AMP and uses small cores only. The ASP metric captures the overall efficiency that a workload derives from the various cores under a particular scheduler.

To illustrate the interrelationship between these two metrics, we carry out a theoretical study on the effectiveness of different scheduling algorithms when running synthetic multi-program workloads on an AMP consisting of two big cores and two small cores. All workloads comprise four CPU-bound single-threaded applications each. In this hypothetical scenario, we assume that applications exhibit constant big-to-small performance ratios that range between 1.0 and 4.7, a similar speedup range to that of the SPEC CPU applications running on the Intel QuickIA prototype we used for our experiments. For single-threaded programs, the speedup matches the *speedup factor* (SF) of its single runnable thread, defined as $\frac{IPS_{big}}{IPS_{small}}$, where IPS_{big} and IPS_{small} are the thread's instructions per second ratios achieved on big and small cores respectively. Each row in Table 1 shows the SFs for individual applications in a workload (W_i).

To approximate fairness and throughput in this scenario we turned to a set of analytical formulas derived in an earlier work [29] and shown in Table 2. As is evident, the formulas only depend on SF_i and F_i, where F_i denotes the big-core time fraction allotted by a given scheduler to application i throughout the execution. The model [29] assumes that (I) $0 \leq F_i \leq 1$, (II) an application's

Table 2
Analytical formulas. Note that the associated performance model [29] does not factor in overheads due to shared-resource contention or thread migrations.

Metric	Analytical formula
Aggregate speedup	$\sum_{i=1}^{n} F_i \cdot (SF_i - 1)$
Unfairness	$\frac{MAX(Slowdown_1,\ldots,Slowdown_n)}{MIN(Slowdown_1,\ldots,Slowdown_n)}$
$Slowdown_i$	$\frac{SF_i}{1-F_i+SF_i \cdot F_i}$

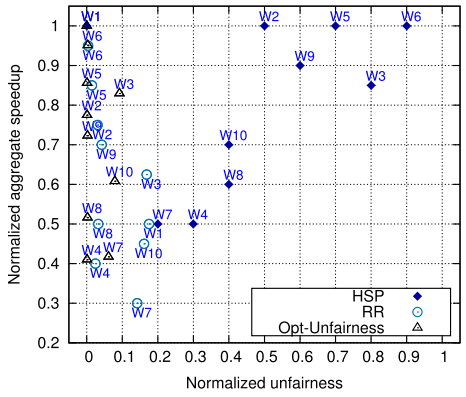


Fig. 1. ASP and unfairness values for the analyzed workloads under the various schedulers. The closer to the top left corner, the better the ASP-Unfairness tradeoff.

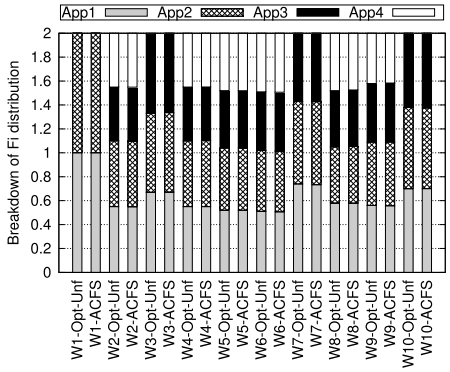


Fig. 2. Big-core cycle distribution among applications for the various workloads in Table 1 under Opt-Unfairness and ACFS.

small-core fraction is $1 - F_i$ (work-conserving scheduler) and (III) $\sum_{j=1}^{n} F_j = N_{BC}$, where N_{BC} denotes the number of big cores in the AMP.

Fig. 1 shows the unfairness and aggregate speedup (ASP) analytical values obtained for the aforementioned workloads under various schedulers. In the figure both metrics have been normalized to the (0,1) interval, where 0 represents the minimum value attainable for the metric in the platform and 1 the maximum value. Specifically, $Norm.\ ASP = \frac{ASP}{(SF_{\max}-1)\cdot N_{BC}}$ and $Norm.\ Unfairness = \frac{Unfairness-1}{SF_{\max}-1}$, where $SF_{\max}$ represents the maximum SF attainable in the AMP. For our study, we used three asymmetry-aware schedulers. The first one, aimed to optimize throughput and denoted as HSP (High-SPeedup) [15,27,13], assigns all big cores to the N_{BC} single-threaded applications in the workload that experience the greatest big-to-small speedup. For these applications $F_i = 1$; the remaining threads are mapped to small cores ($F_i = 0$). The second scheduler is an asymmetry-aware round-robin (RR) policy, a fairness-aware scheme that equally shares big cores among applications [3] (i.e. $\forall j\ F_j = \frac{N_{BC}}{n}$). Finally, the third scheduler, referred to as Opt-Unfairness, constitutes a theoretical algorithm which ensures the maximum ASP value attainable for the optimal unfairness. Determining per-application big-core cycle distributions for this theoretical schedule requires an extensive exploration of the search space. To that end we created a simulator that makes use of the analytical formulas in Table 2 and finds the optimal solution in each case via a branch-and-bound algorithm. Specifically, the search algorithm operates as follows. Given a workload, defined by the set of applications' SFs, the algorithm computes the (Unfairness, ASP) pair for each possible distribution of big-core cycles among the applications. Because exploring the whole continuous search space is unfeasible, candidate solutions are created varying F_i from 0 to 1 in steps of 0.01, such that $\sum F_i = N_{BC}$. In addition, simple heuristics are used to prune unpromising solutions.

Results from Fig. 1 show that HSP optimizes the aggregate speedup (the higher, the better) at the expense of obtaining the worst unfairness numbers (the higher, the worse) by far. The comparison between the HSP scheduler and Opt-Unfairness reveals that, in general, throughput and fairness are largely conflicting optimization goals on AMPs. Clearly, the theoretical Opt-Unfairness scheduler needs to sacrifice much throughput in some cases (up to 22% for W2) to achieve the optimal unfairness. Note that in some workload scenarios (like W3, W7 and W10) it is not possible to deliver the same slowdown across applications, leading to normalized unfairness values slightly bigger than 0. As for RR, which does not consider application speedup when allotting big-core cycles, the results highlight that this policy degrades both fairness and ASP compared to Opt-Unfairness in most cases, thus providing a suboptimal solution.

Prior to evaluating our OS scheduling approach – the ACFS scheduler – on real asymmetric hardware, we implemented it on top of the simulator, so as to compare it against the optimal fairness scheduler (Opt-Unfairness). We found that in the synthetic scenario considered and assuming perfect speedup estimations, the *base* ACFS design (described in Sections 3.1–3.3) performs in less than a 1% range of Opt-Unfairness in terms of fairness and throughput; essentially, both schedulers make almost the same big-core cycle distribution among applications. As an illustrative case, Fig. 2 shows the similarities between the big-core cycle distributions made by both schedulers for workloads in Table 1. Note that we explored with additional workloads and observed the same trends.

2.2. Related work

In discussing related work we begin by covering scheduling proposals that seek to optimize throughput and outline schemes designed to improve fairness. We then recap previous work that focuses on reducing energy consumption.

Throughput optimization and determining the speedup. To maximize throughput in multi-application scenarios, previous research has demonstrated that the scheduler must follow the *HSP approach*, namely it must preferentially run on big cores those applications that derive a higher big-to-small speedup. The main difference between the available variants of the HSP approach [15,3,31,27,13,24] lies in the mechanism employed to obtain threads' speedup factors online. Three techniques have been explored to do so. The first approach comes down to measuring SFs directly [15,3], which entails running each thread on big and small cores to track the IPC (instructions per cycle) on both core types. This approach, known as *IPC sampling*, is subject to inaccuracies associated with program-phase changes [31]. The second approach relies on *estimating a thread's SF* using its runtime properties collected online on any core type using performance counters [13,27,23]. SF estimation requires to derive performance models specifically tailored to the platform in question. The third technique is PIE [35], a hardware-aided mechanism, which has been shown to provide accurate SF estimates. Notably, PIE poses certain shortcomings that could render it difficult its integration on real hardware [24]. In

the implementation of our scheduling proposal (ACFS), we use the second approach to determine a thread's SF online. We elaborate on this aspect in Section 4.

Recent research has highlighted that making scheduling decisions based on per-thread SFs only may lead to serious throughput degradation when multithreaded programs are included in the workload [28,27]. This stems from the fact that the SF does not approximate the overall benefit that a multithreaded application as a whole derives from using the big cores in an AMP [1,10]. Catering to application-wide speedups is the key to optimizing throughput in these workload scenarios. Previous research [27,29] has devised analytical formulas to approximate the speedup for several types of multithreaded applications based on the runnable thread count (a proxy for the amount of thread-level parallelism in the applications), the SF of the application threads and the number of big cores in the AMP. We turned to these formulas to approximate the speedup for multithreaded applications in ACFS's implementation.

Other researchers proposed specific support to accelerate multithreaded programs on AMPs [1,27,11,12,18]. These proposals make use of big cores as accelerators for different types of scalability bottlenecks in parallel applications by employing software [1,27] or hardware-aided approaches [11,12,18]. Our OS-level scheduling proposal is largely orthogonal to these approaches.

Fairness and priority enforcement. 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 performing periodic thread migrations [3]. Fair-sharing big cores has been shown to provide better performance and more repeatable completion times across runs [17] on AMPs compared to default schedulers in general-purpose OSes, which are asymmetry agnostic. RR has been widely used as a baseline for comparison [3,31,27], but as shown in Section 2.1, it constitutes a suboptimal fairness solution.

Li et al. [17] proposed A-DWRR, which aims to deliver fairness on AMPs by factoring in the computational power of the various cores when performing per-thread CPU accounting. To that end, it relies on an extended concept of CPU time for AMPs: *scaled* CPU time. Using scaled CPU time, CPU cycles consumed on a big core are *worth* more than on a small one. To ensure fairness, A-DWRR evens out the scaled CPU time consumed across threads in accordance to their priorities. As opposed to ACFS, A-DWRR does not take into account the fact that applications derive different (and possibly varying) speedups when using big cores in the platform. As our experimental results reveal, this leads A-DWRR to degrading both fairness and throughput.

The Prop-SP scheduler [26] was designed to overcome A-DWRR's main limitations. Prop-SP strives to even out the slowdown experienced by equal-priority applications (fairness), while maintaining acceptable system throughput. To make this possible, each application receives big-core cycles in proportion to the product of its speedup and its priority. Unlike ACFS, Prop-SP is unable to provide a configurable fairness/throughput trade-off and, as our experiments reveal, it is subject to high unfairness in some cases.

In [5] the authors devise an EQual-Progress (EQP) scheduler that seeks to optimize fairness on AMPs. As ACFS, EQP takes per-thread SF values into consideration when tracking the slowdown that each thread in the workload experiences at run time and tries to enforce fairness by evening out observed slowdowns. Nevertheless, EQP and ACFS exhibit important differences. First, when determining a thread's slowdown, EQP does not factor in the past speedup phases the thread underwent. Instead, the slowdown is approximated by taking into account the total cycle count that the thread has consumed on each core type thus far and the current SF [5]. ACFS, on the contrary, maintains a per-thread counter that accumulates the total thread's progress based on the current and the past speedup application phases. In Section 3, we describe this mechanism in detail. Second, EQP was designed to achieve equal slowdown across threads, and so it only takes into account the SF of individual threads when computing slowdowns. ACFS, by contrast, takes into account the application-wide speedup to guarantee *equal slowdowns among applications*. We observed that this feature makes it possible for ACFS to provide a better support when multithreaded applications are included in the workload. Third, ACFS supports user-defined priorities, while the EQP scheduler does not. Note also that EQP relies on either IPC sampling or PIE to obtain SFs online [5]. Since PIE is not available on existing asymmetric hardware, in this work we evaluated the history-based variant of EQP, which is based on IPC sampling.

Optimizing energy consumption. Other researchers have devised ways to reduce energy and power consumption on AMPs [20,14,22,23,36]. Most of these approaches are orthogonal to our fairness proposal.

Petrucci et al. [22] proposed Octopus-Man, a scheme that aims to improve energy efficiency while ensuring that the QoS constraints associated with latency-sensitive jobs are satisfied. This is one of the few works that employs the QuickIA [4] prototype, the same asymmetric system we used to evaluate our scheduling proposal. Nevertheless, the authors do not attempt to devise a mechanism to obtain accurate per-thread big-to-small speedup estimates for such a system, as we do here. In a more recent work, Petrucci et al. [23] propose a scheme to determine the thread-to-core mappings that optimize energy efficiency. As in our proposal, this scheme employs a mechanism to estimate the speedup factor online. Specifically, their approach to estimate the SF leverages a regression model that factors in two performance metrics (MIPS and LLC misses) gathered using hardware counters. We observed that relying on these two performance metrics alone is not sufficient to accurately predict the speedup factor on the Intel QuickIA prototype we used. This issue did not become apparent in [23] since their user-level proposal was evaluated using asymmetric hardware where cores differ in processor frequency only.

3. Design

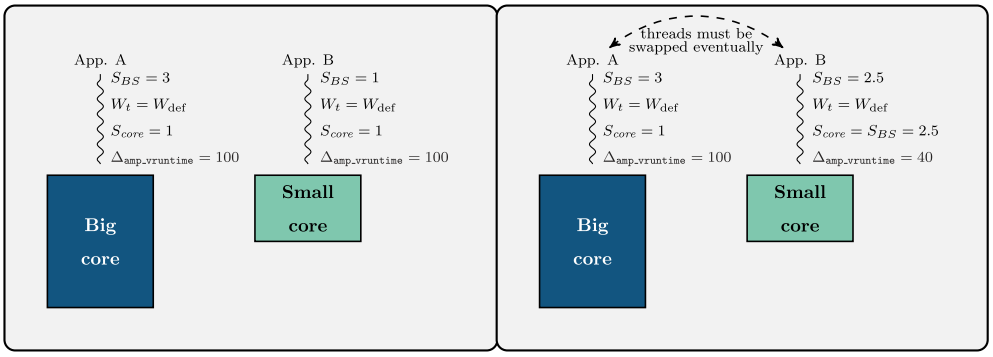
This section describes our scheduling proposal: the ACFS scheduler. We begin by describing the underlying mechanism used by ACFS to track progress as well as the initial core assignment for newly created threads. We then analyze an example in detail to illustrate the intuition behind the mechanism to track progress, and to demonstrate why thread swaps are necessary in some cases to ensure that running applications experience equal slowdown. Finally, we elaborate on the design of the *unfairness factor*, a knob in ACFS enabling the system administrator to trade fairness for throughput.

3.1. Tracking progress and initial assignment

To keep track of applications' relative progress on the AMP, the scheduler assigns each thread a counter called `amp_vruntime`. When a thread runs for a clock *tick* on a given core type, ACFS increments its `amp_vruntime` by $\Delta_{\texttt{amp\_vruntime}}$, which is defined as follows:

$$\Delta_{\texttt{amp\_vruntime}} = \frac{100 \cdot W_{\text{def}}}{S_{core} \cdot W_t} \tag{3}$$

where W_t is the thread's weight, derived directly from the application priority (set by the user); W_{def} is the weight of



(a) Scenario 1. (b) Scenario 2.

Fig. 3. Hypothetical thread-to-core mappings under the ACFS scheduler.

applications with the default priority[1]; and S_{core} is the slowdown experienced by the application for this clock tick (relative to using big cores in the AMP). Hence, when a thread is mapped on the big core, $S_{\text{core}} = 1$ (no slowdown). When it runs on a small one, $S_{\text{core}} = S_{\text{BS}}$, where S_{BS} represents the speedup that the application this thread belongs to would derive from using the big cores in the AMP, relative to using small cores only. Note that the S_{BS} may vary over time as the application goes through different program phases. As such, catering to the varying speedup is essential to accurately track the relative progress throughout the execution. As discussed in Section 2.2, ACFS approximates S_{BS} at run time by factoring in the thread's SF as well as the runnable thread count of the application (a proxy for the amount of thread-level parallelism) using the approach described in our previous work [27,29]. In turn, the thread's SF is estimated by feeding an estimation model with high-level metrics (such as the IPC or the LLC miss rate) collected via hardware counters. We elaborate on this aspect in Section 4.3.

For single-threaded programs, the `amp_vruntime` counter associated with the single-runnable thread indicates how much progress the application has made thus far with respect to the progress that would have resulted from running it on a big core the whole time. The example analyzed in the next section illustrates this fact. In a multithreaded application, tracking the application-wide progress is more complex, because different threads in the application can be mapped to different core types for short periods of time and threads may block sometimes due to synchronization. Instead of maintaining a global per-application progress counter, which may cause contention, ACFS uses the `amp_vruntime` counter of the various threads in the application to ensure fairness. This makes it possible for ACFS to track the relative progress that a given thread has made in the AMP with respect to other threads of the same application. Ensuring equal progress among threads when a multithreaded HPC application goes through a parallel phase is crucial to improve performance [5,11,12]. Moreover, as our experiments reveal (Section 5), using per-thread `amp_vruntime` counters enables the scheduler to deliver system-wide fairness in multi-application scenarios.

When a new thread enters the system, the thread is assigned to the idlest core in the platform. Hence, the load balance across the various cores is preserved. In selecting the target core for a newly created thread, ACFS populates big cores first, since this approach proves effective when it comes to maximizing throughput [16]. Note that the `amp_vruntime` of a newly created thread is set to the maximum `amp_vruntime` value observed among threads in the system. This initial value enables a *fair* comparison between `amp_vruntimes` for threads that entered the system at different points in time.

3.2. Case study

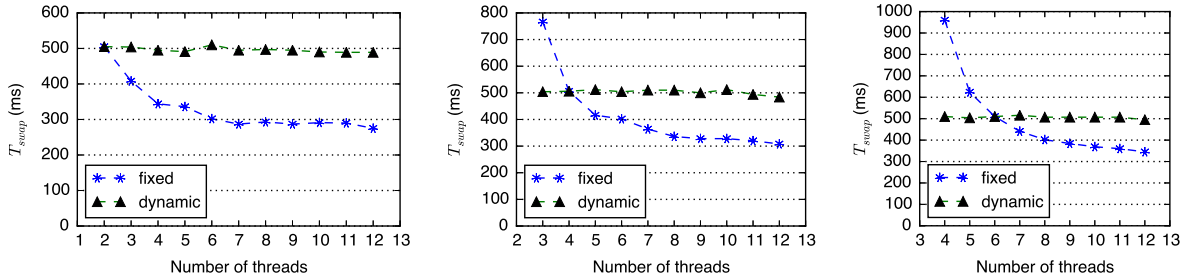
To illustrate the intuition behind ACFS's mechanism for tracking progress (Eq. (3)), let us consider the hypothetical scenario depicted in Fig. 3(a). Two single-threaded applications (*A* and *B*) with the default priority ($W_t = W_{\text{def}}$) run on an AMP system featuring one big and one small core. To maintain load balance, ACFS initially maps *A* to the big core and *B* to the small one. Suppose further that at a certain point in time application *A* is going through a program phase with $S_{\text{BS}} = 3$. In other words, running this application phase on a big core would be three times faster than running it on the small core. In contrast, application *B* runs a program phase that derives no benefit at all from the big core ($S_{\text{BS}} = 1$). As *A* and *B* run, the `amp_vruntime` counter increases by 100 units per tick for both threads (according to Eq. (3)). This increment reflects that both threads are making the same (and maximum attainable) progress (100%) on the AMP system, which leads to similar slowdown (1) for both applications and, in turn, to the optimal unfairness value. Notably, this thread-to-core mapping also ensures optimal throughput, since the application with the highest relative speedup is mapped to the faster core. By keeping track of the progress of both threads, ACFS performs optimally (as Opt-Unfairness) in this context.

Now let us suppose that application *B* enters a new program phase with $S_{\text{BS}} = 2.5$, as depicted in Fig. 3(b). At this point, application *B* would speed up if it were mapped to the big core. If the thread-to-core assignment remains the same, the optimal throughput is still achieved, since the application going through the program phase with the highest relative speedup is mapped to the big core. However, this mapping does not guarantee fairness. Essentially, as *B* runs, its `amp_vruntime` increases by 40 units per tick (*B* makes only 40% of the attainable progress). In contrast, *A*'s `amp_vruntime` increases by 100 units per tick. Under these circumstances, the difference between `amp_vruntimes` will increase over time and so will unfairness in the AMP system.

3.3. Enforcing fairness via thread swaps

To guarantee fairness, ACFS must even out the progress across applications. This comes down to balancing the `amp_vruntime` across threads. To make this possible ACFS may need to perform thread swaps (migrations) between different core types every so often. Note that thread swaps are preferred over *one-way* thread migrations, since swaps do not disturb load balance.

[1] The approach used by ACFS to factor in priorities in CPU accounting is inspired by the mechanism used by Linux's Completely Fair Scheduler, which is described in detail in [19].



(a) 1 big—11 small. (b) 2 big—10 small. (c) 3 big—9 small.

Fig. 4. T_{swap} for different number of threads on various AMP configurations.

Because frequent thread swaps may introduce significant overheads, the scheduler does not trigger a swap as soon as it detects that a thread T_A running on a big core has a greater `amp_vruntime` than that of a thread T_B running on a small core. Instead, the ACFS scheduler swaps T_A and T_B in the event that ($\texttt{amp\_vruntime}_{T_A} - \texttt{amp\_vruntime}_{T_B}$) exceeds a given configurable threshold. Increasing this threshold makes it possible to effectively reduce the thread-swap frequency thus mitigating migration overheads. Conversely, very high threshold values may lead to accumulating unfairness for longer periods.

Notably, using a fixed threshold does not guarantee a uniform average thread-swap period (T_{swap}) for different number of threads over a certain period of time (*Total_time*); specifically $T_{swap} = \frac{Total\_time * N_{BC}}{Total\_swaps}$, where N_{BC} denotes the number of big cores in the AMP. To enforce a target value of T_{swap}, the threshold must be properly adjusted by factoring in the system load as well as N_{BC}. Making the system administrator responsible for setting the threshold manually based on the system load can be a big burden. Moreover, constantly changing this threshold by hand can be largely impractical in workload scenarios where the number of runnable threads varies very frequently. To overcome these shortcomings, the ACFS scheduler exposes a configurable parameter, referred to as `amp_threshold`, which represents a *base* value of the threshold. This base value enforces a target value of T_{swap} when the total number of threads is twice the number of big cores. At run time, the actual threshold used by ACFS is automatically adjusted by multiplying `amp_threshold` by the following factor[2]: $\frac{2 \cdot (N_T - N_{BC})}{N_T}$, where N_T denotes the total number of threads. Intuitively, when the number of threads does not exceed the number of big cores, ACFS does not trigger migrations, so the scale factor is not taken into consideration in this scenario. In practice, the scale factor ensures that the actual threshold used by ACFS is smaller than `amp_threshold` when $N_{BC} < N_T < 2 \cdot N_{BC}$. By contrast, when $N_T > 2 \cdot N_{BC}$ the actual threshold is bigger than `amp_threshold`.

To achieve a target average swap period (T_{swap}), the value of the base threshold (`amp_threshold`) can be approximated as follows:

$$\texttt{amp\_threshold} = 100 \cdot \frac{T_{swap}}{2 \cdot T_{tick}} \cdot \left(1 - \frac{1}{SF_{avg}}\right) \tag{4}$$

where SF_{avg} denotes the average speedup factor observed in the platform, and T_{tick} represents the *tick* length. We derived this formula analytically by calculating how long it takes for two sequential applications (with $W_t = W_{\text{def}}$) that have just been swapped to be swapped again in a scenario where $N_T = 2 \cdot N_{BC}$.

To demonstrate that adjusting the threshold dynamically in this way guarantees a target swap period, we performed a sensitivity study using various 12-core AMP configurations. (These configurations are based on the AMD platform described in Section 4.1.) Specifically, on each AMP configuration we ran different number of instances (ranging from $N_{BC} + 1$ to the total core count) of a single-threaded application. For all the experiments we set a constant value of the `amp_threshold` value so as to enforce an average swap period of 500 ms. Fig. 4 shows the actual average thread-swap period for different number of threads when using a dynamically adjusted threshold, and a fixed one (equal to `amp_threshold`). As is evident, the dynamically adjusted threshold enables ACFS to approximate the target average swap period, whereas using a constant threshold only guarantees the target period in the base case ($N_T = 2 \cdot N_{BC}$). In practice, the scale factor used effectively enlarges the threshold as the number of threads increases so as to approximate to the target swap period.

3.4. Throughput-fairness trade-off

As illustrated in Section 2.1, the *base* ACFS described thus far closely tracks the behavior of the Opt-Unfairness theoretical scheduler, which provides the maximum throughput attainable for the optimal unfairness. Because fairness and throughput are clearly conflicting objectives (as shown in our theoretical study), Opt-Unfairness may degrade the system throughput significantly at the expense of delivering the optimal (minimal) unfairness value. To overcome this limitation under the ACFS scheduler, we created the `unfairness_factor` (UF) knob, which makes it possible for the system administrator to trade fairness for throughput in scenarios where fairness constraints are relaxed.

When the UF knob is set at its default value (1.0), the scheduler behaves as the base implementation, hence attempting to achieve the maximum throughput attainable for the optimal unfairness. For *UF* values >1, ACFS increases throughput at the expense of degrading fairness up to a certain extent. Intuitively, making this possible comes down to gradually increasing the big-core share for those applications in the workload with a higher speedup while reducing the big-core share of the remaining ones. The main challenge is how to gradually improve throughput while keeping fairness under control. To this end, we factor in the *UF* when updating a thread's `amp_vruntime` every tick. This entails replacing the thread's *static* priority or weight (W_t) in Eq. (3) with its *dynamic* weight (DW_t), which is defined as follows:

$$DW_t = W_t \cdot \left(1 + \frac{(\text{UF} - 1) \cdot (S_{BS} - S_{\min})}{S_{\max} - S_{\min}}\right) \tag{5}$$

where $S_{\max}$ and $S_{\min}$ are the maximum and minimum speedups (S_{BS}s) observed among applications in the workload, respectively. Essentially, by replacing the thread's static weight (W_t) with its dynamic counterpart (DW_t), the `amp_vruntime` of high-speedup threads is incremented at a slower pace than that of low-speedup

[2] This factor was determined empirically by analyzing how the average swap period varies with the number of threads while maintaining a constant value for the threshold under different AMP configurations. In oversubscription scenarios, the factor is downscaled by also taking the average run queue length into consideration.

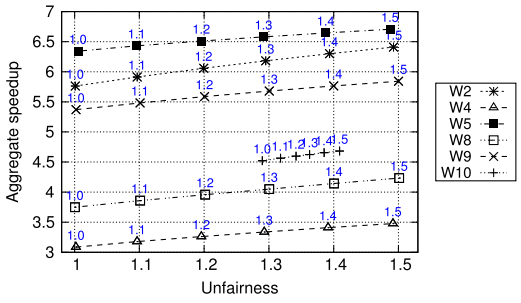


Fig. 5. Theoretical unfairness and aggregate speedup for different UF values. The optimal unfairness value for workload W10 is 1.29; the optimal value for the other workloads is 1.

threads, which results in a higher big-core share for high-speedup applications and, in turn, in higher system throughput.

Using our implementation of ACFS on top of the simulator described in Section 2.1, we observed that gradually increasing the `unfairness_factor` (UF) for a workload under ACFS leads to throughput gains while ensuring unfairness no greater than $UF \cdot opt$, where *opt* denotes the optimal unfairness for the workload in question. Fig. 5 showcases this trend by showing the analytical aggregate speedup (throughput) and unfairness values for some synthetic workloads from Table 1 and different UF settings (labeled in the figure). It should be noted that the theoretical fairness results shown in Fig. 5 can only be reached with perfect speedup estimates. In Section 5.4, we analyze the effect of varying the UF using our real-world implementation of ACFS in the Linux kernel.

4. Determining the speedup factor

As explained in the previous section, the ACFS scheduler takes a thread's speedup factor (SF) into consideration when measuring the thread's progress on the AMP. To determine a thread's SF online, ACFS feeds a platform-specific estimation model with values from diverse performance metrics collected over time. Because the value of a performance metric (e.g., IPC) may differ across core types, the scheduler relies on two models: one for predicting the SF from big-core metrics and another for predicting it from small-core metrics. These two estimation models (used for all threads) are generated offline. It should be noted that building accurate estimation models is a challenging task, since it requires the identification of a reduced subset of performance metrics that enable to explain the performance differences that come from running a thread on various core types [13]. The fact that these metrics are largely architecture-dependent and vary with the underlying form of performance asymmetry of the AMP, further complicates the identification of these metrics [27].

The remainder of this section is organized as follows. Section 4.1 introduces our experimental setting, which is crucial to understand the different forms of performance asymmetry we explored, as well as the challenges that arise when building SF estimation models on various platforms. Section 4.2 presents our proposed methodology to generate these models via offline processing. Finally, Section 4.3 describes how our implementation of the ACFS scheduler in the Linux kernel uses the estimation models at run time.

4.1. Experimental setup

To evaluate the effectiveness of ACFS we experimented with the Intel QuickIA prototype system [4] as well as with an AMD-based multicore server platform.

The Intel QuickIA platform consists of a dual-socket UMA system featuring two multicore processors: a quad-core Intel Xeon E5450 processor and a dual-core Intel Atom N330 processor. To reduce shared-resource contention effects[3] for the experiments on this system we disabled one core on each die in the Xeon processor. This setting gives us a pairing of two high-performance *big* cores (E5450) with two low-power *small* ones (N330). We will refer to this asymmetric configuration as 2B-2S. The different cores in this platform feature three fixed-function performance monitoring counters (PMCs) and two general-purpose (configurable) PMCs. In practice, this allows the gathering of three insightful performance metrics simultaneously on any core type: the IPC, using fixed-function PMCs, and two other high-level metrics using general-purpose PMCs. As we show later, this restriction coupled with the profound differences across cores makes SF estimation on this platform a difficult task.

Because of the reduced core count on the Intel QuickIA, we opted to explore other AMP configurations more suitable to run workloads including multithreaded programs. Specifically, we also experimented with a NUMA multicore server that integrates two AMD Opteron 2425 hex-core processors. On this platform we emulated an AMP configuration consisting of 2 big cores and 10 small ones (2B-10S) by reducing the processor frequency of some cores. Specifically, "big" cores on 2B-10S operate at 2.1 GHz, whereas "small" cores run at 800 MHz. Each core on the platform is equipped with four configurable PMCs.

4.2. Generating speedup factor estimation models offline

To aid in the construction of platform-specific SF estimation models, we first opted to apply the methodology presented as part of our earlier work [27], which at a high level, entails performing the following steps:

1. Select a representative set *AP* of sequential applications as well as a comprehensive set of performance metrics *M*, allowing to characterize the microarchitectural and memory behavior of the various applications.
2. Run applications in *AP* on both core types to obtain their overall speedup factor (i.e., ratio of completion times on both core types) and gather the values for performance metrics in *M* using PMCs. Because PMCs are a limited resource, this step may entail running the applications more than once in some platforms.
3. Build estimation models to approximate the SF from the big and from the small core as follows. Using the overall SF as well as the average metric values collected for the entire execution of applications in *AP* on both core types, apply additive regression [7] to approximate the SF from the metric values. For this task we turned to the additive-regression engine implemented in the WEKA machine-learning tool [9].
4. From the big-core and the small-core model obtained in the previous step, identify the subset of metrics ($SM \subseteq M$) with greater regression coefficients that can be monitored simultaneously in the platform in question. Specifically, gathering the values for metrics in *SM* must not require more performance counters than those available in the platform. This makes it possible to avoid in-kernel event multiplexing, when estimating the SF.
5. Finally, build the final big-core and small-core SF estimation models by performing the same actions as in Step 3 but using only the collected values for metrics in *SM*. In doing so, the final estimation models just depend upon metrics in *SM*.

[3] None of the scheduling algorithms analyzed in this work deals with share-resource contention issues. We leave for future work analyzing the effects of coupling performance asymmetry with serious shared resource contention scenarios.

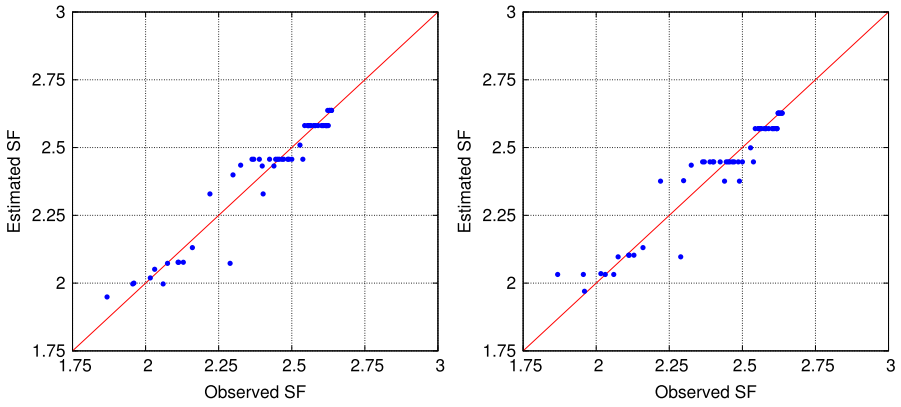


Fig. 6. SF prediction on the big core (left) and the small core (right) on the AMD system obtained with *Overall-SF*. Perfectly accurate estimations have all points on the diagonal line.

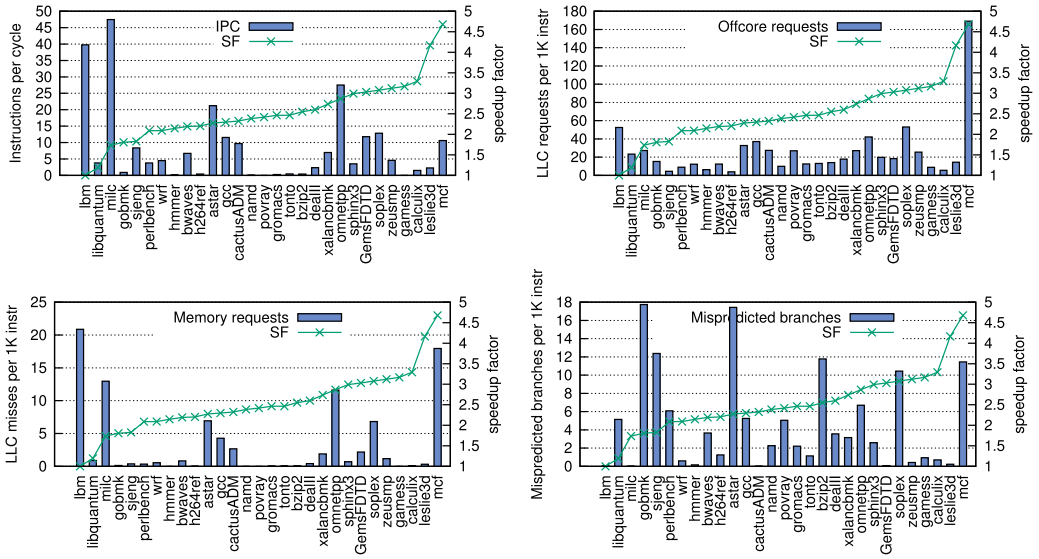


Fig. 7. Speedup factor vs. several big-core performance metrics for SPEC CPU 2006 benchmarks running on the Intel QuickIA prototype.

Henceforth, we will refer to this methodology as the *Overall-SF* approach. To build SF estimation models in our experimental platforms using this technique, we employed some applications in the SPEC CPU2006 and CPU2000 suites as the *AP* set. For these applications, we monitored a wide range of high-level performance metrics, such as the IPC, LLC (Last-level-cache) misses and LLC requests per 1K instructions, the branch misprediction rate, the number of DTLB and ITLB misses per 1M instructions, etc.

Fig. 6 shows the observed and predicted SFs on the big and the small core of the AMD system. On this platform, *Overall-SF* yields to accurate and simple estimation models, which rely primarily on memory and cache-related metrics. The simplicity of this model stems from the fact that cores in this AMP setting differ only in processor frequency; in this scenario the LLC miss or the LLC access rates are known to show a negative correlation with the SF [13,27]. Specifically, these models achieve correlation coefficients of 0.97 and 0.96 when predicting the SF on the big and the small core. Despite the fact that these models were generated using SPEC CPU benchmarks, we also experimented with additional applications from other benchmark suites, as shown in Section 5.

On the QuickIA prototype we could not obtain accurate SF estimation models by using *Overall-SF*. The more pronounced differences between cores (microarchitecture, cache sizes, processor frequency, etc.) coupled with the reduced set of high-level metrics that can be monitored simultaneously at run time, make it harder to estimate the SF online on this system. With such a restriction, SF estimation models that depend upon three performance metrics achieve correlation coefficients no higher than 0.80. On this platform, we also observed that none of the performance metrics alone we gathered for the entire execution of the applications exhibits a clear correlation with the average speedup factor. (Fig. 7 shows the trends for some of these metrics.) This makes it difficult for the additive-regression engine to generate accurate models from this data.

We also analyzed the effectiveness of SF estimation models that depend on more than three performance metrics on the Intel QuickIA. Note that, to use these performance models at run time, the scheduler must perform event multiplexing (i.e., different sets of performance metrics must be monitored in a round-robin fashion). Event multiplexing makes it possible to eliminate the restriction on the small number of metrics that can be monitored, and allows us to obtain estimation models with a correlation coefficient equal to 0.90 on both core types. Fig. 8 depicts the observed and predicted SFs with these models on the big and the small core. Despite the 0.90 correlation coefficient for the overall SFs (average throughout the execution), we found that these models do not provide accurate estimations for individual program phases in the applications. Essentially, on the Intel QuickIA prototype some applications exhibit coarse-grained phases with an SF value greater than the maximum overall SF observed for the SPEC CPU benchmarks (4.7 in this platform). For example, as Fig. 9(a) reveals, the `mcf` benchmark exhibits long-term phases

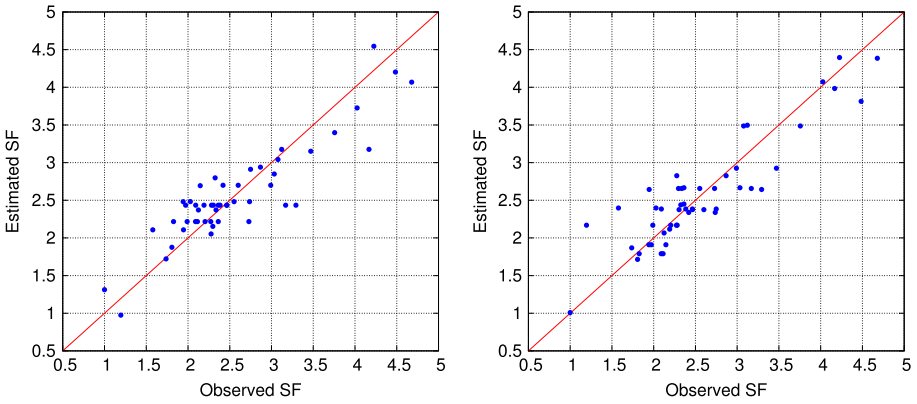


Fig. 8. SF prediction on the big core (left) and the small core (right) on the Intel system for the SPEC CPU benchmarks obtained with *Overall-SF*.

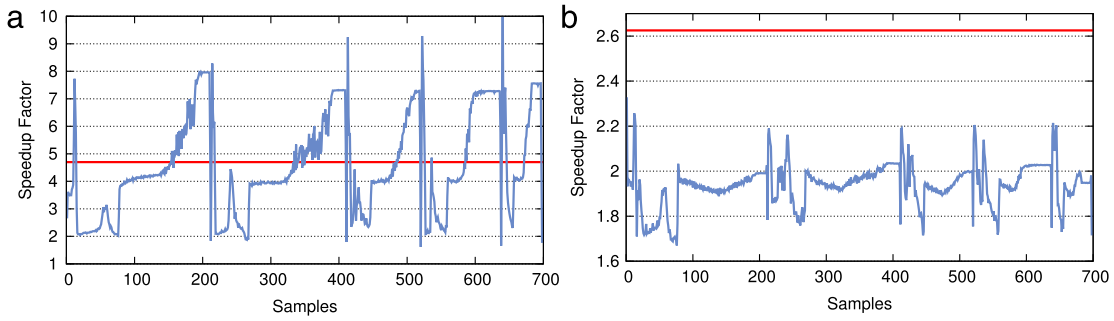


Fig. 9. SF over time for `mcf` on the Intel (a) and the AMD (b) systems. The horizontal red line indicates the maximum overall SF observed for SPEC CPU applications on each platform. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

with an SF value as high as 8. Unfortunately, the generated models provide poor estimations for such high-SF program phases. By contrast, models built for the AMD system are not subject to this issue, and provide accurate estimations for different program phases. Note that benchmarks on the AMD platform do not exhibit coarse-grained phases with an SF value greater than the maximum overall SF observed (2.625). As shown in Fig. 9(b), the `mcf` benchmark does not go through long phases with an SF greater than 2.625 on such a system.

To generate SF models with a higher estimation accuracy on the Intel QuickIA, we devised a variant of *Overall-SF*. Instead of using the overall SF and metric values for the entire execution of the benchmarks to generate the models, the new approach leverages information associated with *different program phases*. We will refer to this new scheme as *Phase-SF*. To track the SF and performance-metric values for different program phases in a single-threaded application we sample performance counters every 200 million instructions on both core types. We observed that gathering offline information by using this sampling rate makes it possible to effectively capture coarse-grained program phases and filter out many SF spikes. Note that the SF of a certain application's instruction window is the ratio of the IPS values (big-to-small) collected on both core types for that instruction window.

From the SF trace for the various instruction windows of an application, we identify coarse-grained program SF phases. The mechanism to break down an SF trace into phases is described in the Appendix. Once coarse-grained SF phases have been detected for an application, we generate a compact summary of each SF phase, which consists of a tuple including the geometric mean of the values for each performance metric as well as the geometric mean of the SF for the various samples belonging to the same program phase. Finally, we use the collection of per-application SF-phase summaries obtained in the previous step as input to the additive-regression engine. This makes it possible to generate the final SF models for both core types.

Fig. 10 shows the SFs predicted with the final estimation models on the Intel QuickIA obtained by means of *Phase-SF*. The correlation coefficients for the estimation on the big core and the small core are 0.95 and 0.94 respectively. In generating the models, we used performance data from 742 SF phases from SPEC CPU benchmarks. Notably, we found that using information from 500 diverse SF phases is enough to generate a big-core model with a similar accuracy to that of the best model we found for the Intel QuickIA. By contrast, obtaining a small-core model with a correlation coefficient no smaller than 0.94 requires monitoring data from at least 700 SF phases. Despite the fact that the additive-regression prediction engine we used enabled us to discard some metrics from the final estimation models, the models still depend upon more than three performance metrics on both core types (shown in Table 3), so the scheduler implementation needs to turn to event multiplexing to estimate the SF online in this case. We elaborate on this issue in the next section.

To conclude the discussion, it is worth noting that the offline analysis required to build the speedup factor estimation models has to be performed just once on each asymmetric platform. Thus, to deploy the mechanism transparently on a production system, the associated experiments could be automatically launched by the OS when it boots for the first time. Alternatively, estimation models can be automatically rebuilt periodically by leveraging idle system periods to collect new performance samples for additional applications. In our experiments, we carried out a full execution of a subset of benchmarks from SPEC CPU on the different core types of the system, in order to perform a comprehensive characterization of the performance of these benchmarks on each evaluated AMP configuration. This can be a time-consuming process, especially on the *small* core of the QuickIA platform (Intel Atom), so it may not constitute a practical approach. Nevertheless, we found that a complete execution of the benchmarks is not really necessary to build accurate estimation models. In practice,

Table 3
Performance metrics used to predict the SF on the Intel QuickIA system.

Performance metrics	Used in big-core model	Used in small-core model
Instructions retired per cycle (IPC)	✓	✓
L2 cache accesses per 1K retired instr.	✓	✓
L2 cache misses per 1K retired instr.	✓	✓
Branch instructions retired per 1K instr.	✓	✓
Mispredicted branches per 1K retired instr.	✓	✓
ITLB misses per 1M retired instr.		✓
DTLB misses per 1M retired instr.		✓

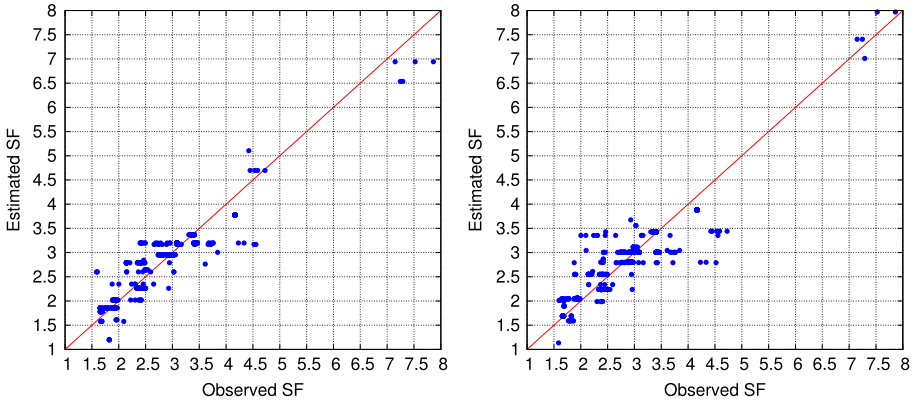


Fig. 10. SF prediction on the big core (left) and the small core (right) for different phases from SPEC CPU benchmarks on the Intel QuickIA. Models were obtained using *Phase-SF*.

the vast majority of the representative program phases come from a few short phased benchmarks, such as `soplex`, `astar` or `gcc`. Many other relevant phases become apparent at the beginning of the execution of long-running benchmarks, such as `lbm` or `libquantum`, which exhibit a largely regular behavior. Hence, gathering information for just a small number of instruction windows from a set of representative benchmarks suffices to obtain models with similar accuracy; this constitutes a more viable option.

4.3. Implementation: using SF-estimation models at run time

To implement the estimation models for the AMD platform and the Intel QuickIA we created two independent monitoring modules using the PMCTrack monitoring tool [25]. The monitoring modules (deployed in a loadable kernel module) feed the scheduler with per-thread SF estimates at run time. In using this approach, the core scheduler implementation (inside the OS kernel) remains fully architecture independent and completely decoupled from the underlying technique to obtain SFs online [25]. To obtain SF estimates, the monitoring module in question continuously gathers the necessary performance metric values via hardware counters and applies the estimation model. In our setting, the counters are sampled on a per-thread basis every 200 ms; this sampling interval was used in previous work on scheduling on AMPs [23,27]. We observed that the overhead associated with sampling and SFs estimation is negligible at this rate. To experiment noticeable overhead (around 1%) on our experimental platforms, the sampling period has to be reduced to a value as low as 8 ms.

Unlike SF estimation on the AMD platform, determining SFs on the Intel QuickIA requires monitoring a set of performance metrics whose values cannot be gathered simultaneously using the number of performance counters available. Thus, on the QuickIA, gathering the information necessary for the estimation model entails monitoring different sets of hardware events in a round-robin fashion (aka. event multiplexing). On this platform, two PMC sampling periods are required for the big-core model and three for the small-core model. Once all the required performance metrics have been gathered on a certain core type using the necessary sampling periods, the metric values are used to predict the thread's SF. This process is repeated continuously throughout the thread's execution to feed the scheduler with up-to-date SF estimates.

We found that an important issue arises when implementing such a scheme on the Intel QuickIA. Performance metrics monitored in subsequent PMC sampling periods on a given core may not belong to the same program phase, and using event metric values from different phases leads to inaccurate SF predictions. To overcome this issue, we augmented the prediction scheme with a heuristic to detect transitions between program phases. This heuristic is similar to that proposed in [3], which was evaluated in a simulation environment. At a high level, the heuristic works as follows. For each thread, we maintain a running average of the IPC. To this end, we always monitor the number of instructions per cycle together with the other metrics in a particular event set. If a sudden variation of the IPC is detected in the last sample (with respect to the running average of the IPC), we assume that the sample belongs to a new program phase. If a sample for a different program phase is detected, previously collected samples from the same sampling round are discarded when estimating the SF, and a new sampling round is started. By contrast, if an entire sampling round is completed without detecting phase transitions, then the samples collected are used to generate an up-to-date SF value with the associated estimation model. We observed that this heuristic turns out effective in detecting phase transitions, and, in turn, leads to more accurate SF estimates over time.

5. Experimental evaluation

In our experiments, we carry out an extensive comparison of ACFS with several state-of-the-art schedulers for AMPs: HSP [13,27], Prop-SP [26], the history-based variant of EQP [5], RR [3] and A-DWRR [17]. We implemented all scheduling algorithms as a separate scheduling class in the Linux kernel v3.2. Like ACFS, the HSP, Prop-SP and EQP schemes make scheduling decisions by taking thread speedup factors (SFs) into consideration, so they rely

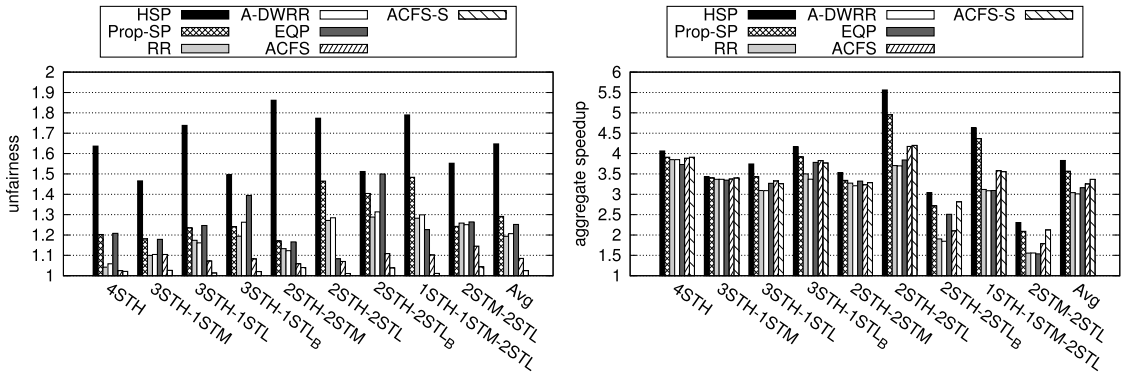


Fig. 11. Fairness and throughput on 2B-2S (Intel QuickIA).

on hardware performance counters to function. As stated in [5], the history-based variant of EQP employs IPC sampling to determine SFs online. We found that the HSP and Prop-SP schedulers yield better performance when using estimation models to determine a thread's SF online. As such, in our final experiments, we opted to use SF estimation rather than IPC sampling for these schedulers.

Our evaluation targets multi-application workloads consisting of benchmarks from diverse suites (SPEC CPU2006, OMP 2001, PARSEC and Minebench). We also experimented with BLAST – a bioinformatics benchmark; and FFTW3D – a program performing the fast Fourier transform. In all the experiments, the total thread count in the workload was set to match the number of cores in the platform, as in previous work on AMPs [13,27,23] that also employs CPU-bound workloads. In multi-application experiments, we ensure that all applications are started simultaneously and when an application terminates it is restarted repeatedly until the longest application in the set completes three times. We then obtain the aggregate speedup and unfairness for the scheduler in question, by using the geometric mean of the completion times for each program.

In the remainder of this section we analyze the results for four sets of experiments. First, we discuss the effectiveness of the various schedulers for multi-application workloads consisting of applications with the same priority. Second, we analyze multi-application scenarios where applications are assigned different priorities. Third, we show the effects of adjusting ACFS's `unfairness_factor` knob. Finally, we carry out a sensitivity study that showcases the impact of the thread swap frequency on throughput and fairness under ACFS.

5.1. Applications with the same priority

In creating the workloads, we categorized applications into three groups with respect to their parallelism: highly parallel (HP), partially sequential (PS) – parallel programs with a serial component of over 25% of the total execution time – and single threaded (ST). We further divided the three aforementioned application groups into three subclasses based on their SFs—high (H), medium (M) and low (L). The program mixes we explored mimic scenarios with different SF ranges and varying degree of competition for the scarce big cores in the AMP.

(A) *Workloads consisting of single-threaded applications.* Fig. 11 shows the results for workloads in Table 4 running on 2B-2S (Intel QuickIA). These program mixes include single-threaded applications only. The workload name shown in the table encodes the category of each application as it appears in the corresponding row. For example, in the 2STH-2STL workload, `mcf` and `calculix` are STH (Single-Threaded High-SF) programs and `sjeng06` and `gobmk` are STL (Single-Threaded Low-SF) applications. Note that the SF subcategories (H, M and L) are largely architecture specific.

Table 4
Multi-application workloads consisting of single-threaded applications.

Workload	Benchmarks
4STH	`calculix`, `gamess`, `GemsFDTD`, `bzip2`
3STH-1STM	`calculix`, `GemsFDTD`, `bzip2`, `h264ref`
3STH-1STL	`gamess`, `GemsFDTD`, `bzip2`, `sjeng`
3STH-1STL$_B$	`calculix`, `gamess`, `sphinx3`, `sjeng`
2STH-2STM	`gamess`, `soplex`, `povray`, `h264ref`
2STH-2STL	`mcf`, `calculix`, `sjeng`, `gobmk`
2STH-2STL$_B$	`gamess`, `sphinx3`, `gobmk`, `libquantum`
1STH-1STM-2STL	`mcf`, `h264ref`, `sjeng`, `gobmk`
2STM-2STL	`namd`, `h264ref`, `gobmk`, `libquantum`

For example, the `gobmk` benchmark is classified as a low-SF program on 2B-2S (Intel QuickIA) but it constitutes a high-SF program on 2B-10S (AMD platform).

As shown in Fig. 11, the scheduler that optimizes throughput (HSP) effectively obtains the best aggregate speedup, but that comes at the expense of delivering the worst unfairness numbers (the higher, the worse) across the board. As for RR and A-DWRR, both schemes fair-share big cores among threads in this scenario, and so they perform similarly in most cases. As is evident, fair-sharing big cores may lead to throughput and fairness degradation, especially for workloads exhibiting a wide range of big-small speedups (e.g., 3STH-1STL and 2STH-2STL). In addition, the results highlight that Prop-SP is able to achieve a better balance between throughput and fairness than that of HSP. However, Prop-SP is still subject to high unfairness in some cases (e.g., 2STH-2STL).

Now we zoom in on the results for EQP and ACFS, which strive to optimize fairness. Clearly, EQP is not able to obtain lower unfairness than RR or A-DWRR for all workloads, thus failing to achieve its main goal. We found that this primarily stems from the inaccuracies associated with the mechanism employed by EQP to approximate threads' slowdowns, as pointed out in Section 2.2. Moreover, EQP is subject to high SF mispredictions that come from its reliance on IPC sampling on off-the-shelf AMPs. IPC sampling has been shown to lead to inaccurate SFs, since IPC values collected on each core type may belong to different program phases [31]. We observed that these inaccurate values become apparent especially in the last four workloads. In contrast, the ACFS scheduler is not subject to the aforementioned program-phase issues, since it predicts a thread's SF by means of an estimation model that uses performance metrics collected on the *current core type*. The results reveal that, despite the existing imperfections in the SF model, ACFS is able to obtain the best unfairness figures across the board. On average it reduces unfairness by 10% compared to RR and A-DWRR, and by 13% compared to EQP, while ensuring better system throughput than these schemes. To evaluate the impact of SF inaccuracies on ACFS's behavior, we also experimented with a static version of ACFS (ACFS-S), where the scheduler is fed with applications' SFs measured offline for the entire execution.

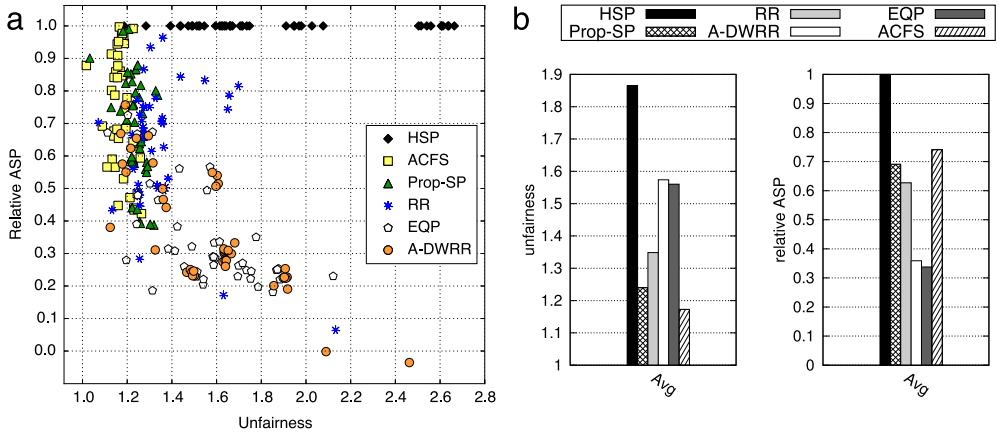


Fig. 12. Results for workloads consisting of single-threaded and multithreaded applications.

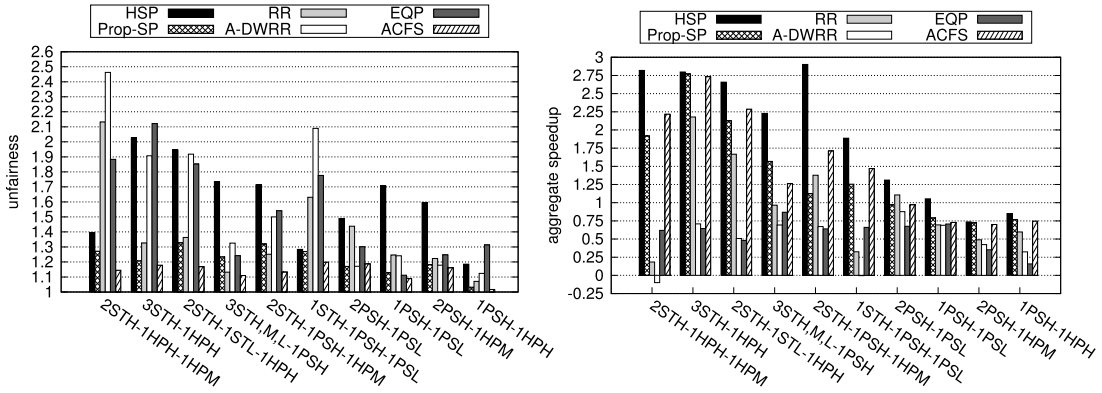


Fig. 13. Fairness and throughput for selected workloads on 2B-10S (AMD platform).

As evident, perfect overall SF values enable to reduce unfairness even further. This fact underscores that high accuracy in speedup estimation is paramount when it comes to delivering fairness on AMPs.

(B) *Workloads consisting of single-threaded and multithreaded applications.* Due to the reduced core count of the 2B-2S configuration, we used the 2B-10S (AMD platform) configuration to experiment with workloads consisting of parallel and sequential applications. In this workload scenario there is a much bigger set of workload types to explore, since there are 9 potential application categories: 3 parallelism classes (ST, HP and PS) with 3 SF subclasses each (H, M and L). To cater to this higher diversity, we generated 44 random program mixes by combining 14 applications from different categories. In generating the workloads we made sure that at least one multithreaded application was included in each program mix. Overall, we observed that these workloads exhibit a wider big-to-small speedup range across applications than that of workloads consisting of single-threaded applications only. Essentially, many workloads combine parallel applications that derive a modest speedup from using the scarce big cores (such as HP applications) with other applications that experience significant performance gains when using these cores (such as single-threaded applications).

Fig. 12(a) shows the unfairness vs. throughput for all the workloads considered. For the sake of clarity, all the throughput (ASP) values displayed are normalized with respect to the throughput delivered by the HSP scheduler. Note that the closer a (workload) point is to the top left corner, the better the throughput-fairness tradeoff. To illustrate how the various algorithms perform in general, Fig. 12(b) displays the average unfairness and relative ASP values observed across workloads. In addition, to gain further insight into the results, Fig. 13 displays the individual unfairness and throughput values for 10 selected workloads from different categories that represent very different points in Fig. 12(a). Table 5 shows the composition of these workloads. Note that for multithreaded applications, the number in parentheses shown by each program's name in the table is the number of threads it runs with.

Table 5
Subset of workloads consisting of single-threaded and multithreaded applications.

Workload	Benchmarks
2STH-1HPH-1HPM	`gamess`, `hmmer`, `fma3d_m`(5), `wupwise_m`(5)
3STH-1HPH	`hmmer`, `gobmk`, `h264ref`, `fma3d_m`(9)
2STH-1STL-1HPH	`perlbench`, `soplex`, `h264ref`, `fma3d_m`(9)
3ST{H, M, L}-1PSH	`gamess`, `astar`, `soplex`, `blackscholes`(9)
2STH-1PSH-1HPM	`hmmer`, `perlbench`, `semphy`(5), `wupwise_m`(5)
1STH-1PSH-1PSL	`gobmk`, `BLAST`(6), `FFTW3D`(5)
2PSH-1PSL	`BLAST`(4), `semphy`(4), `FFTW3D`(4)
1PSH-1PSL	`semphy`(6), `FFTW3D`(6)
2PSH-1HPM	`BLAST`(4), `semphy`(4), `wupwise_m`(4)
1PSH-1HPH	`BLAST`(6), `fma3d_m`(6)

For HSP and ACFS, results in Figs. 12 and 13 showcase similar trends as those of workloads on 2B-2S: ACFS achieves the best fairness figures across the board while HSP obtains slightly better throughput than ACFS at the expense of degrading fairness significantly. Notably, we observe that ACFS obtains more modest fairness improvements over Prop-SP (5.5% on average) than those observed for workloads including single-threaded programs only (16%). In this case, we found that Prop-SP makes a big-core cycle distribution more similar to that of ACFS, hence the similar fairness and throughput results. Specifically, assigning big-core cycles in proportion to the application speedup (Prop-SP) is known to provide better fairness results when both multithreaded and single-threaded applications are included in the workload [29].

Table 6
Average reduction in unfairness and increase in throughput achieved by ACFS over the other schemes across the whole set of explored workloads.

ACFS vs. others	Reduct. in Unf.	Increase in ASP
HSP	36.65%	−24.16%
Prop-SP	7.25%	2.78%
RR	12.94%	15.80%
A-DWRR	23.39%	76.10%
EQP	23.21%	73.65%

While the results associated with the ACFS, Prop-SP and HSP schedulers exhibit a consistent and clear trend across the board, we observe important divergences in the results of EQP and A-DWRR. Numbers in Fig. 13 enable us to explain the source of the major divergences. The results reveal significant differences between EQP, A-DWRR and the other fairness-aware schedulers especially for workloads including both single-threaded and multithreaded applications (the first six workloads in Table 5). More specifically, EQP and A-DWRR perform very poorly (in terms of both fairness and throughput) for workloads that combine sequential programs with highly parallel applications (such as 3STH-1HPH or 2STH-1HPH-1HPM). In fact, workload points concentrated in the bottom center of Fig. 12(a) for these two schedulers correspond to workloads with such a composition. The poor fairness and throughput numbers in this context stem from the fact that EQP and A-DWRR make scheduling decisions without taking into consideration the number of runnable threads in the various applications (a proxy for the amount of thread-level parallelism). Specifically, A-DWRR ensures that each thread in the workload receives the same AMP-scaled CPU time, regardless the application it belongs to. EQP aims to enforce equal slowdown across threads by considering the SF of individual threads only, but ignoring the number of threads in the associated applications. Under these two schedulers, the higher the number of threads in the application, the higher the big-core share allotted to the application. Because highly-parallel applications are known to derive lower speedup from the scarce big cores than applications with limited thread-level parallelism [27,1], favoring highly-parallel applications over single-threaded programs leads A-DWRR and EQP to worse unfairness and throughput than the other schemes. These results enable us to draw a very important conclusion: enforcing fairness across individual threads in the system without factoring in the application thread count (what A-DWRR and EQP do) does not ensure *equal slowdowns among applications* when multithreaded programs are included in the workload. Notably, Prop-SP and ACFS do take the number of threads in the application into consideration when making scheduling decisions, which leads to fairness and throughput benefits in this context. Specifically, this enables ACFS to reduce unfairness by 25% on average relative to EQP and A-DWRR, and it yields significantly higher throughput.

For workloads that do not include any sequential application (the last four workloads in Table 5), EQP and A-DWRR provide unfairness and throughput values closer to those of ACFS and Prop-SP. Note that multithreaded applications used in these workloads derive modest speedup values from using the scarce fast cores. Low speedup values across applications (observed when each program runs alone on the AMP) typically lead to low per-application slowdown values in the workloads. This results in smaller unfairness and throughput figures for all fairness-aware schedulers in this scenario, including the RR scheduler. This scheduling scheme, however, outperforms EQP and A-DWRR for workloads that combine one HP and several single-threaded applications. In that context, RR grants a higher big-core share to threads from single-threaded programs than to individual threads in HP applications. As stated earlier, this is not the case under EQP and A-DWRR, which leads these two schedulers to higher unfairness and throughput degradation than RR. Despite this fact, ACFS still reduces unfairness by 13% on average with respect to RR.

(C) *Overall results.* Table 6 shows the observed average reduction in unfairness and average increase in throughput achieved by ACFS over the other schemes across the whole set of workloads that we explored −9 of them made up of single-threaded programs, plus 44 mixes consisting of single-threaded and multithreaded applications. As is evident, ACFS reduces unfairness substantially over the other fairness-aware schemes: 23% with respect to EQP and A-DWRR, 13% compared to RR, and 7% over Prop-SP. Moreover, our approach is able to deliver substantial throughput (ASP) gains compared to these schemes.

5.2. Applications with different priorities

We now assess the effectiveness of the schemes that support user-defined priorities (A-DWRR, Prop-SP and ACFS) in scenarios where applications with different priorities coexist on the 2B-2S configuration. Specifically, we experimented with a workload featuring two high-SF applications (`gamess` and `bzip2`), one mid-SF program (`h264ref`) and a low-SF one (`gobmk`). Such a workload enables us to explore how different applications can be accelerated as the priority increases and how unfairness is affected under the various schedulers.

Our experiments consist in gradually increasing the priority of one selected *high-priority application* (HPA) while keeping the priority of the remaining applications at the default setting. For each selected HPA, we gradually increase its priority so that the associated weight (W_t) increases in steps of 25%. Fig. 14 shows the results. The x-axis indicates the selected HPA; the associated weight W_t, derived directly from the application priority, is specified in parentheses. For the different priority settings and schedulers, we report the HPA's relative speedup and the unfairness. To factor in application priorities in the unfairness metric (as in [6]), we replaced slowdowns in Eq. (1) with their weighted counterparts: $W_t \cdot Slowdown_{app}$. Note also that the HPA speedup is normalized to A-DWRR in the scenario when all applications have the same priority or weight. This enables us to track by how much the HPA speeds up with the priority. Since we observed very similar trends for the two single-threaded high-speedup programs, we omitted the results for `bzip2`.

In the scenario where all applications have the same priority (labels with the "(1.0)" suffix), A-DWRR fair-shares big cores among all applications. In contrast, Prop-SP and ACFS grant a higher big-core share to `gamess`, `bzip2` and `h264ref`; these applications derive a greater speedup from using big cores than `gobmk`. As the HPA's priority increases, all schedulers are able to reduce its completion time, since they all gradually increase the HPA's big-core share with the priority. As is evident, only ACFS is able to maintain low unfairness as the HPA's priority increases; Prop-SP and A-DWRR are subject to higher fairness degradation. Throughput results, omitted due to space constraints, reflect similar trends to those observed in scenarios with equal-priority applications.

5.3. Trading fairness for throughput under ACFS

Recall that the ACFS scheduler is equipped with the `unfairness_factor` (UF) knob, which empowers the user with a means to provide a configurable balance between fairness and the system throughput extracted from the AMP. Fig. 15 shows how the UF choice affects fairness and throughput for three selected workloads from Table 4 running on the Intel QuickIA. Both metrics have been normalized to the (0,1) interval, by using the mechanism

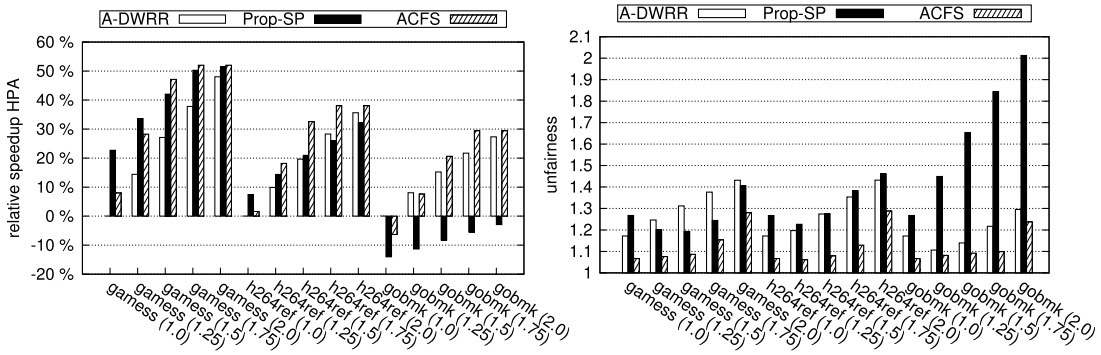


Fig. 14. HPA speedup (left) and unfairness (right) for program mixes consisting of applications with different priorities.

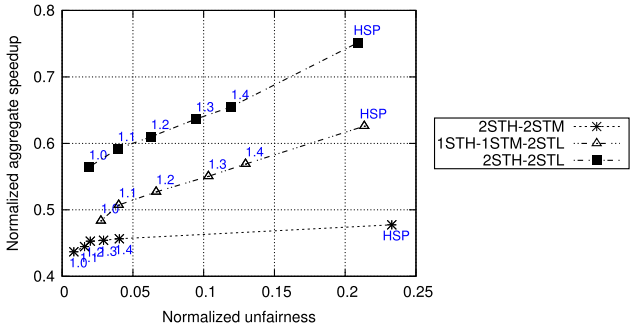


Fig. 15. Normalized unfairness vs. normalized aggregate speedup for different UF values.

described in Section 2.1. The results reveal that the default and lowest possible setting for the UF (1.0) provides the best fairness figures, while higher UF values always lead to throughput gains at the expense of degrading fairness. Notably, the trends illustrate that, by gradually increasing the UF, the ACFS scheduler can get closer to the HSP scheduler, which optimizes throughput.

5.4. Impact of the thread swap frequency on throughput and fairness

All the scheduling algorithms considered in our study trigger thread swaps every so often to accomplish different goals. Our implementations of the different algorithms in the Linux kernel rely on the same mechanism for swapping threads. As an illustrative case, we analyze the impact of the thread swap frequency on throughput and fairness under ACFS. For our analysis we used 3 program mixes with four sequential programs each running on two AMP configurations (on the Intel and on the AMD platform) consisting of 2 big cores and 2 small cores.

It is well known that thread migrations introduce both software and hardware overhead [16]. The software overhead includes the time to move one thread from one core to another; this entails acquiring multiple run queue locks. Note that in the scenario considered (one thread per core) the two migrations in a thread swap must be often serialized since, in addition to acquiring the locks, the scheduler must trigger the preemption of two running threads from different CPUs. Serializing migrations in a swap may lead to load imbalance for very short periods of time. As for the hardware overhead associated with a migration, we must consider two aspects: (1) the thread has to rebuild its cache state (up to three cache levels on the AMD platform), which involves extra cache misses; and (2) the cost of cache misses after a migration can be higher on NUMA platforms (AMD system) than on UMA machines (Intel QuickIA) [16].

Fig. 16 shows the impact on unfairness and aggregate speedup that comes from varying the average swap period from 100 ms to 1 s. The results reveal that the throughput degradation and the unfairness clearly decrease when increasing the swap period. (In the experiments of the previous sections we set the average swap period to 850 ms, as this setting provides satisfactory results across platforms, as shown in the figure.) For the first workload, which consists of four instances of `gamess`, we observe that the unfairness stays the same even for small values of the swap period. This stems from the fact that the `gamess` program is CPU intensive and has a small working set; such a program experiences negligible hardware overhead after migration. Nevertheless, setting the average swap period to a value lower than 300 ms leads to non-negligible software overhead – primarily related to the transient load imbalance situations – which has a negative impact on throughput. The other two workloads combine three instances of `gamess` with a mildly memory-intensive application (`astar`) and a highly memory-intensive program (`libquantum`), respectively. These memory-intensive applications experience a significant hardware-related overhead when migrated frequently, as opposed to gamess. This leads to uneven application slowdown and hence to higher unfairness for small values of the swap period.

6. Conclusions

In this paper we proposed ACFS, a scheduler that seeks to optimize fairness on Asymmetric single-ISA Multicore Processors (AMPs) while maintaining acceptable system throughput. To make this possible, ACFS evens out the slowdown that the various applications in the workload experience as a result of sharing the AMP system. To track the relative progress that an application makes over time, the scheduler takes into account the relative benefit (speedup) that the application derives from using the big cores in the AMP as it goes through different program phases. Our experimental evaluation, using real hardware and scheduler implementations in the Linux kernel, reveals that ACFS is able to reduce unfairness by 23% on average compared to two state-of-the-art fairness-aware schemes [17,5] and by 13% relative to an asymmetry-aware round-robin policy [3,27]. Moreover, ACFS yields considerably higher system throughput than these approaches. Notably, ACFS's throughput and fairness gains relative to the other schedulers are especially pronounced for workloads including multithreaded programs. We also demonstrated that previous scheduling proposals that support user priorities on AMPs [3,26] are subject to high fairness degradation in the event that applications with different priorities coexist on the system; ACFS, by contrast, ensures low unfairness in this context.

Key elements for the success of ACFS are the mechanism used to keep track of the relative progress made by each application on the AMP, and its reliance on online estimation models to approximate

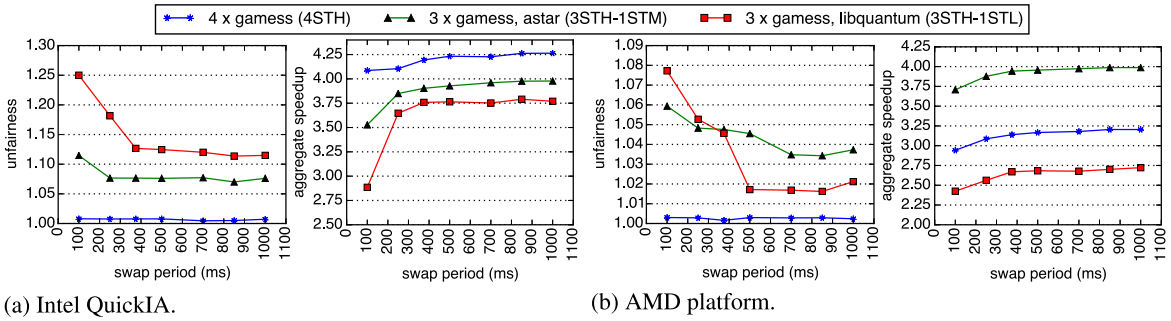


Fig. 16. Unfairness and aggregate speedup when varying the average swap period under ACFS on AMP configurations consisting of 2 big cores and 2 small cores.

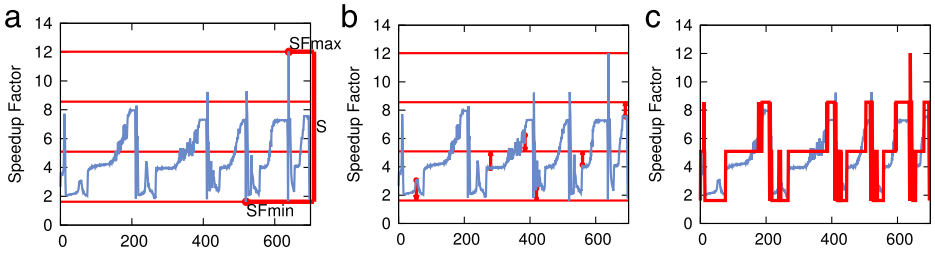


Fig. 17. Breaking down the SF trace of the `mcf` program into phases: (a) SF subintervals for $U = 3$, (b) Nearest threshold value, and (c) Resulting SF phases.

the relative benefit that a thread derives from running on a big core relative to a small core (aka speedup factor). One of the major challenges that became apparent when implementing ACFS as well as other schedulers used for comparison purposes, was to devise estimation models based on performance counters to predict a thread's speedup factor at run time on highly asymmetric hardware. Existing methodologies to aid in generating those estimation models have proven effective on AMP platforms where cores differ in processor frequency only or exhibit slight microarchitectural differences [27]. We found that these schemes turn out ineffective for asymmetric systems such as the Intel QuickIA prototype [4] used for our experiments, where cores exhibit profound differences (microarchitecture, frequency, cache sizes, etc.). In this work, we proposed a methodology enabling to derive accurate speedup-factor prediction models for such a system.

Acknowledgments

This work has been supported by the EU (FEDER) and the Spanish MINECO, under grant TIN 2015-65277-R, as well as by the HIPEAC-4 European Network of Excellence. We would like to thank David Koufaty (Circuits and Systems Research Lab at Intel) and Alexandra Fedorova (University of British Columbia) for enabling us to experiment with the QuickIA prototype system. We also thank the anonymous reviewers for their valuable comments and suggestions.

Appendix

This appendix describes the mechanism used to break down a speedup factor (SF) trace for the entire execution of a single-threaded application into coarse-grained SF phases. Recall that the *Phase-SF* methodology (presented in Section 4.2) relies on the ability to identify distinct phases in an SF trace. We define a phase as a set of contiguous instruction windows where either the SF remains constant or does not vary significantly across the various samples. Several schemes have been proposed to identify program phases at run time by employing either hardware-based [34,33] or software approaches that can be exploited by the OS scheduler [3]. Here, we aim to detect these phases offline from an SF trace of the entire execution of the application collected beforehand.

Our scheme to break down an SF trace into phases is inspired by thresholding algorithms. It entails performing the following four steps. First, find the maximum and minimum SF values ($SF_{\max}$ and $SF_{\min}$) in the SF trace. Second, divide the observed SF range into U same-sized subintervals, where U is a configurable parameter. Let S be $SF_{\max} - SF_{\min}$, and let p be the length of each subinterval ($p = S/U$). Hence, each ith SF subinterval ($i \in \{0 \,..\, U-1\}$) is defined as $[SF_{\min} + p \cdot i, SF_{\min} + p \cdot (i+1)]$. Fig. 17(a) depicts a potential division into subintervals for the SF trace of the SPEC CPU `mcf` benchmark running on the Intel QuickIA. Third, for each sample in the SF trace, find the nearest threshold value (limit of a subinterval). Specifically, given a certain SF value, such that $u \leq SF \leq u + p$, where u and $u + p$ are the limits of a subinterval, the nearest threshold value (NTV) function is defined as follows:

$$NTV(SF) = \begin{cases} u & \text{if } (\text{SF} - u) \leq (u + p - \text{SF}) \\ u + p & \text{if } (\text{SF} - u) > (u + p - \text{SF}) \,. \end{cases} \tag{6}$$

Fig. 17(b) illustrates how the NTV function is applied to the SF trace of the `mcf` benchmark. The last step of the process consists in breaking down the SF trace into phases using the information obtained in the previous step, as shown in Fig. 17(c). Essentially, two consecutive SF samples SF_j, SF_{j+1} in the trace are said to belong to the same phase if $NTV\left(SF_j\right) = NTV\left(SF_{j+1}\right)$.

Despite the simplicity of the scheme, it allows us to detect coarse-grained phases relatively well for the SPEC CPU benchmarks. Because the most appropriate choice for the U parameter is application specific, we employed an iterative algorithm that finds the lowest value for this parameter such that the average deviation in the resulting SF phases reaches a certain threshold. Note also that, since our goal is to factor in coarse-grained phases when creating the SF estimation model, phases with scarce SF samples (e.g., spikes in the figures) are discarded when applying the *Phase-SF* methodology (Section 4.2).

References

[1] M. Annavaram, E. Grochowski, J. Shen, Mitigating Amdahl's Law through EPI Throttling, in: Proceedings of International Symposium on Computer Architecture, ISCA'05, Wisconsin, USA, 2005, pp. 298–309.

[2] ARM, Benefits of the big.LITTLE Architecture, 2013. https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf (accessed: 31.10.16).

[3] M. Becchi, P. Crowley, Dynamic thread assignment on heterogeneous multiprocessor architectures, in: Proceedings of International Conference on Computing Frontiers, CF'06, Ischia, Italy, 2006, pp. 29–40.

[4] N. Chitlur, G. Srinivasa, S. Hahn, P.K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, R. Iyer, QuickIA: Exploring heterogeneous architectures on real prototypes, in: Proceedings of International Conference of High Performance Computer Architecture, HPCA'12, New Orleans, LA, 2012, pp. 1–8.

[5] K.V. Craeynest, S. Akram, W. Heirman, A. Jaleel, L. Eeckhout, Fairness-aware scheduling on single-ISA heterogeneous multi-cores, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT'13, Edinburgh, Scotland, 2013, pp. 177–187.

[6] E. Ebrahimi, C.J. Lee, O. Mutlu, Y.N. Patt, Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'10, Pittsburgh, PA, 2010, pp. 335–346.

[7] J.H. Friedman, Stochastic gradient boosting, Comput. Statist. Data Anal. 38 (4) (2002) 367–378.

[8] R. Gabor, S. Weiss, A. Mendelson, Fairness and throughput in switch on event multithreading, in: Proceedings of International Symposium on Microarchitecture, MICRO'06, Orlando, FL, 2006, pp. 149–160.

[9] M.A. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update, ACM SIGKDD Explor. Newslett. 11 (2009) 10–18.

[10] M.D. Hill, M.R. Marty, Amdahl's law in the multicore era, IEEE Comput. 41 (7) (2008) 33–38.

[11] J.A. Joao, M.A. Suleman, O. Mutlu, Y.N. Patt, Bottleneck identification and scheduling in multithreaded applications, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'12, London, England, UK, 2012, pp. 223–234.

[12] J.A. Joao, M.A. Suleman, O. Mutlu, Y.N. Patt, Utility-based acceleration of multithreaded applications on asymmetric cmps, in: Proceedings of the International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, 2013, pp. 154–165.

[13] D.A. Koufaty, D. Reddy, S. Hahn, Bias scheduling in heterogeneous multi-core architectures, in: Proceedings of the European Conference on Computer systems, EuroSys'10, Paris, France, 2010, pp. 125–138.

[14] V. Kumar, A. Fedorova, Towards better performance per watt in virtual environments on asymmetric single-ISA multi-core systems, SIGOPS Oper. Syst. Rev. 43 (3) (2009) 105–109.

[15] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, K.I. Farkas, Single-ISA heterogeneous multi-core architectures for multithreaded workload performance, in: Proceedings of the International Symposium on Computer Architecture, ISCA'04, Munchen, Germany, 2004, p. 64–.

[16] T. Li, D. Baumberger, D.A. Koufaty, S. Hahn, Efficient operating system scheduling for performance-asymmetric multi-core architectures, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC'07, 2007, pp. 53:1–53:11.

[17] T. Li, P. Brett, R.C. Knauerhase, D.A. Koufaty, D. Reddy, S. Hahn, Operating system support for overlapping-ISA heterogeneous multi-core architectures, in: Proceedings of the International Conference on High-Performance Computer Architecture, HPCA'10, Bangalore, India, 2010, pp. 1–12.

[18] N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, A. Cristal, Thread lock section-aware scheduling on asymmetric single-ISA multi core, IEEE Comput. Archit. Lett. 14 (2) (2015) 160–163.

[19] W. Mauerer, Professional Linux Kernel Architecture, Wrox Press Ltd., Birmingham, UK, UK, 2008.

[20] J.C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, V. Talwar, Using asymmetric single-isa cmps to save energy on operating systems, IEEE Micro 28 (3) (2008) 26–41.

[21] O. Mutlu, T. Moscibroda, Stall-time fair memory access scheduling for chip multiprocessors, in: Proceedings of International Symposium on Microarchitecture, MICRO'07, Chicago, IL, 2007, pp. 146–160.

[22] V. Petrucci, M.A. Laurenzano, J. Doherty, Y. Zhang, D. Mossé, J. Mars, L. Tang, Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers, in: Proceedings of the International Symposium on High Performance Computer Architecture, HPCA'15, San Francisco, CA, 2015, pp. 246–258.

[23] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N.A. Gazala, S. Gobriel, Energy-efficient thread assignment optimization for heterogeneous multicore systems, ACM Trans. Embedded Comput. Syst. 14 (1) (2015) 15:1–15:26.

[24] M. Pricopi, T.S. Muthukaruppan, V. Venkataramani, T. Mitra, S. Vishin, Power-performance modeling on asymmetric multi-cores, in: Proc. of International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES'13, Montreal, Canada, 2013, pp. 15:1–15:10.

[25] J.C. Saez, J. Casas, A. Serrano, R. Rodríguez-Rodríguez, F. Castro, D. Chaver, M. Prieto-Matias, An OS-oriented performance monitoring tool for multicore systems, in: Proceedings of International European Conference on Parallel and Distributed Computing, Euro-Par'15: Parallel Processing Workshops, Vienna, Austria, 2015, pp. 697–709.

[26] J.C. Saez, F. Castro, D. Chaver, M. Prieto, Delivering fairness and priority enforcement on asymmetric multicore systems via OS scheduling, in: Proceedings of the International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS'13, Pittsburgh, PA, USA, 2013, pp. 343–344.

[27] J.C. Saez, A. Fedorova, D. Koufaty, M. Prieto, Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems, ACM Trans. Comput. Syst. 30 (2) (2012) 6:1–6:38.

[28] J.C. Saez, A. Fedorova, M. Prieto, H. Vegas, Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors, in: Procedings of the ACM International Conference on Computing Frontiers, CF'10, Bertinoro, Italy, 2010, pp. 31–40.

[29] J.C. Saez, A. Pousa, F. Castro, D. Chaver, M. Prieto-Matías, Exploring the throughput-fairness trade-off on asymmetric multicore systems, in: Proceedings of International European Conference on Parallel and Distributed Computing, Euro-Par'14: Parallel Processing Workshops, Part II, Porto, Portugal, 2014, pp. 326–337.

[30] J.C. Saez, A. Pousa, F. Castro, D. Chaver, M. Prieto-Matias, ACFS: A completely fair scheduler for asymmetric single-ISA multicore systems, in: Proceedings of the ACM Symposium on Applied Computing, SAC'15, Salamanca, Spain, 2015, pp. 2027–2032.

[31] J.C. Saez, D. Shelepov, A. Fedorova, M. Prieto, Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems, J. Parallel Distrib. Comput. 71 (2011) 114–131.

[32] D. Shelepov, J.C. Saez, S. Jeffery, A. Fedorova, N. Perez, Z.F. Huang, S. Blagodurov, V. Kumar, HASS: a scheduler for heterogeneous multicore systems, Oper. Syst. Rev. 43 (2) (2009) 66–75.

[33] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, B. Calder, Discovering and exploiting program phases, IEEE Micro 23 (6) (2003) 84–93.

[34] T. Sherwood, S. Sair, B. Calder, Phase tracking and prediction, in: Proceedings of International Symposium on Computer Architecture, ISCA'03, San Diego, CA, 2003, pp. 336–349.

[35] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, J. Emer, Scheduling heterogeneous multi-cores through performance impact estimation (PIE), in: Proceedings of International Symposium on Computer Architecture, ISCA'12, Portland, OR, 2012, pp. 213–224.

[36] Y. Zhang, L. Duan, B. Li, L. Peng, S. Sadagopan, Cross-architecture prediction based scheduling for energy efficient execution on single-isa heterogeneous chip-multiprocessors, Microprocess. Microsyst. 39 (45) (2015) 271–285.

Juan Carlos Saez received his Ph.D. in computer science from the Complutense University of Madrid (UCM) in 2011. He is now an Associate Professor in the Department of Computer Architecture at UCM. Since 2012 he has been teaching different subjects related to Operating Systems and Computer Architecture. His research interests include energy-aware computing and improving the interaction between the operating system and hardware for emerging architectures. His recent research activities focus on OS scheduling on heterogeneous multicore processors, exploring new techniques to deliver better performance per watt, and quality of service on these systems.

Adrian Pousa received the degree in Computer Science and the Specialization Degree in Networking and Security from the University of La Plata, Argentina, in 2007 and 2012 respectively, where he is currently an Associate Professor. He has been teaching different subjects related to Computer Science since 2005. He is member of the Institute of Research in Computer Science (III-LIDI). His current research interests include: (1) parallel architectures (2) parallel computing techniques (3) OS scheduling techniques for asymmetric multiprocessors.

Fernando Castro obtained the M.S. degree in physics from University of Santiago de Compostela (Spain) in 2000, the M.S. degree in electrical engineering and the Ph.D. degree in computer science from the University Complutense of Madrid (UCM) in 2004 and 2008, respectively. He is now an assistant professor in the Department of Computer Architecture, UCM. His research interests include energy-aware processor design, efficient memory management and OS scheduling on asymmetric multiprocessors.

Daniel Chaver received the degree in physics from the University of Santiago de Compostela, Spain, in 1998, and the Electrical Engineering and Ph.D. degrees from the Complutense University of Madrid, Spain, in 2000 and 2006, where he is currently an Associate Professor. He has been teaching different subjects related to Computer Science and Electrical Engineering since 2000. His current research interests include: (1) architectural techniques for managing efficiently the memory hierarchy and (2) OS scheduling techniques for asymmetric multiprocessors.

Manuel Prieto-Matias received the Ph.D. degree from the Complutense University of Madrid (UCM) in 2000. He is now associate professor in the Department of Computer Architecture at UCM and serves as vice-dean for External Relations and Research at the School of Computer Science and Engineering. His research interests include areas of parallel computing and computer architecture. Most of his activities have focused on leveraging parallel computing platforms and on complexity-effective micro-architecture design. 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 system software and the underlying architecture. He has co-written numerous articles in journals and for international conferences in the field of parallel computing and computer architecture.