ACFS: A Completely Fair Scheduler for Asymmetric Single-ISA Multicore Systems

Juan Carlos Saez
Complutense University
Madrid, Spain
jcsaezal@ucm.es

Adrian Pousa
II-LIDI, UNLP
La Plata, Argentina
apousa@lidi.info.unlp.edu.ar

Fernando Castro
Complutense University
Madrid, Spain
fcastror@ucm.es

Daniel Chaver
Complutense University
Madrid, Spain
dani02@ucm.es

Manuel Prieto-Matias
Complutense University
Madrid, Spain
mpmatias@ucm.es

ABSTRACT

Single-ISA (instruction set architecture) asymmetric multicore processors (AMPs) were shown to deliver higher performance per watt and area than symmetric CMPs (Chip Multi-Processors) for applications with diverse architectural requirements. A large body of work has demonstrated that this potential of AMP systems can be realizable via OS scheduling. Yet, existing schedulers that seek to deliver fairness on AMPs do not ensure that equal-priority applications experience the same slowdown when sharing the system. Moreover, most of these schemes are also subject to high throughput degradation and fail to effectively deal with user priorities.

In this work 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 on a general-purpose OS, demonstrates that ACFS achieves an average 11% fairness improvement over state-of-the-art schemes, while providing better system throughput.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management–Scheduling

Keywords

Asymmetric multicore, scheduling, operating systems, CFS

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 [7]. 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 [7]. The ARM big.LITTLE processor [2] and the Intel Quick-IA prototype [4] demonstrate that major hardware players are following this approach.

Despite their benefits, AMPs pose significant challenges to the system software. One of the main challenges is to efficiently distribute big-core cycles among the various applications running on the system. This task can be accomplished by the OS scheduler [13, 10, 6]. Most existing scheduling schemes have focused on maximizing the system throughput [7, 13, 10, 6]. To this end, the scheduler needs to map to big cores those applications that use these cores efficiently, since they derive performance improvements (speedup) relative to running on small cores [3, 7].

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 [8, 13]. Second, the end user naturally expects that applications with equal priorities get equal slowdowns as a result of sharing the platform. This is not usually the observed behavior in AMPs if the scheduler only seeks to maximize throughput [12]. Third, in scenarios with imposed QoS constraints trading-off 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, 8, 12, 14]. In this work, we demonstrate that existing fairness schemes are either subjected to high throughput degradation or do not constitute effective priority-based schemes. Notably, many of these techniques do not exploit the fact that applications in a multiprogram workload may derive different benefit from using the big cores in the AMP. As a result, they constitute suboptimal fairness solutions [12, 14].

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[1] and evaluated it using real asymmetric hardware. Notably, our proposal effectively enforces user priorities and does not require

[1]We implemented ACFS on top of the asymmetry-agnostic Completely Fair Scheduler (CFS).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695714

hardware support nor changes in the applications. Our experimental evaluation reveals that ACFS improves fairness by 11% compared to state-of-the-art schedulers [8, 14, 12] and at the same time improves throughput by 5%.

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 showcases our experimental results and Section 5 concludes.

2. BACKGROUND AND RELATED WORK

In this section we first discuss the interrelationship between fairness and system throughput on AMPs. We then cover scheduling proposals that seek to optimize throughput and outline those schemes designed to improve fairness.

2.1 Throughput and fairness on AMPs

To quantify throughput on AMPs, previous work [12] has employed the *Aggregate SPeedup* (ASP) metric:

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

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

Previous research on fairness for CMPs [5] and AMPs [14, 12] define a scheme as fair if equal-priority applications in a multi-program workload suffer the same slowdown due to sharing the system. To cope with this notion of fairness, we turned to the lower-is-better *unfairness* metric [5]:

$$Unfairness = \frac{MAX(Slowdown_1, ..., Slowdown_n)}{MIN(Slowdown_1, ..., Slowdown_n)} \quad (2)$$

where $Slowdown_i = CT_{sched,i}/CT_{fast,i}$, and $CT_{fast,i}$ is the completion time of application i when running alone in the AMP (with all the big cores available).

In [12] the authors build a theoretical model to approximate fairness and throughput for different scheduling algorithms, and carried out a study that demonstrates that fairness and system throughput constitute conflicting objectives on AMPs. Specifically, the study considers two asymmetry-aware schedulers. The former, denoted as HSP (High-SPeedup) [7, 6], optimizes throughput by devoting big cores to run applications in the workload that experience the greatest big-to-small speedups. The latter, referred to as Opt-Unfairness, constitutes a theoretical algorithm which ensures the maximum throughput (ASP) attainable for the optimal unfairness. The analysis reveals that HSP optimizes the aggregate speedup at the expense of degrading fairness significantly. Conversely, Opt-Unfairness usually needs to sacrifice throughput to achieve the optimal unfairness.

2.2 Throughput optimization and 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 benefit or *speedup* from big cores. Note that 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.

The main difference between the available variants of the HSP approach [7, 3, 6, 10, 13, 11] lies in the mechanism employed to obtain threads' speedup factors online. Three techniques have been explored to do so. The first approach boils down to measure SFs directly [7, 3], which entails running each thread on big and small cores to track the IPC (instructions per cycle) on both core types. Previous work has demonstrated that this approach, known as *IPC sampling*, is subjected to inaccuracies in SF estimation associated with program-phase changes [13]. The second approach relies on *estimating a thread's SF* using its runtime properties collected on any core type at runtime using performance counters [6, 10, 11]. The third technique is PIE [15], a hardware-aided mechanism enabling accurate SF estimation from any core type. Notably, the required hardware support for PIE has not yet been adopted in commercial systems, and so scheduler implementations on existing asymmetric hardware, as the ones we considered in this work, cannot benefit from this approach. In addition, recent research has highlighted that PIE poses several problems that make it difficult to be deployed on actual hardware [9].

Due to the limitations of IPC sampling and PIE, we opted to use the second approach when implementing ACFS, which also factors in thread's SFs when making scheduling decisions. Specifically, to determine a thread's SF online, ACFS feeds an estimation model with values from diverse performance metrics collected over time[2] (such as the IPC or the last-level-cache miss rate). In this work, we used a variant of the technique proposed in [11] to aid in the construction of platform-specific SF estimation models.

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 [11]. 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, 11]. Catering to application-wide speedups is the key to optimize throughput in these workload scenarios [1, 10, 12]. Previous research [10, 12] 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.

2.3 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 [3]. Fair-sharing big cores has been shown to provide better performance than that of default schedulers on general-purpose OSes, which are asymmetry agnostic, and also provides more repeatable completion times across runs [8]. For this reason, RR has been widely used as a baseline for comparison [3, 10, 11]. Notably, recent research has shown that RR constitutes a suboptimal fairness solution [12], as it does not consider

[2]In our setting, performance counters are sampled per-CPU every 200ms. We observed that the overhead associated with sampling and SFs estimation becomes negligible at this rate.

application speedups when making scheduling decisions.

Li et. al [8] 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 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 [12] 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, is subjected to high unfairness in some cases.

In [14] the authors devise an EQual-Progress (EQP) scheduler that seeks to optimize fairness on AMPs. As ACFS, EQP continuously tracks the slowdown that each thread in the workload experiences at runtime and tries to enforce fairness by evening out observed slowdowns. We found that the EQP scheduler poses several limitations. 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 number of cycles that the thread has consumed on each core type thus far and the current SF[3]. 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. We observed that ensuring that each thread in the system experiences a similar slowdown does not guarantee *equal slowdowns among applications* when multithreaded programs are included in the workload. Third, the EQP scheduler does not support user priorities. Our proposal, described in the next section, overcomes these limitations.

3. DESIGN

ACFS assigns threads to big and small cores so as to preserve load balance in the AMP, and periodically migrates threads between cores to ensure that running applications experience equal slowdown. To keep track of applications' slowdowns, the scheduler assigns each thread a counter called `amp_vruntime`. For single-threaded programs, the counter associated with the single-runnable thread tracks 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. For multithreaded applications, each thread's `amp_vruntime` represents the relative progress that the thread has made in the AMP with respect to other threads of the same application. When a thread runs for a clock *tick* on a given core type, ACFS increments the thread' `amp_vruntime` by $\Delta_{\texttt{amp\_vruntime}}$, defined as follows:

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

where W_t is the thread's weight, derived directly from the application priority (set by the user); W_{def} is the weight of applications with the default priority; S_{core} is the slowdown factor. Note that when a thread runs on the big core, $S_{\text{core}} = 1$. 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 derives from using the big cores on the AMP, relative to using small cores only. As explained in Section 2.2, S_{BS} is estimated at runtime by ACFS for both single-threaded and multithreaded programs taking into account the application's current runtime characteristics. Since S_{BS} may vary over time as the application goes through different program phases, catering to the varying speedup is essential to accurately track the relative progress throughout the execution.

To illustrate the intuition behind Eq. 3 let us consider two single-threaded applications (A and B) running on an AMP with one big and one small core. Suppose further that both applications have the default priority ($W_{\text{def}} = W_t$) and run twice as fast on a big core than on a small one ($S_{\text{BS}} = 2$). To maintain load balance, ACFS could initially map A to the big core and B to the small one. In this scenario, as A runs, its `amp_vruntime` increases by 100 units per tick. In contrast, B's `amp_vruntime` increases by 50 units per tick only (according to Eq. 3), which reflects that A makes twice the progress of B. Clearly, if the thread-to-core mapping does not change, the difference between `amp_vruntimes` will increase over time and so will unfairness in the AMP system.

ACFS aims to enforce fairness by evening out the `amp_vruntime` across threads[4]. To make this possible it may need to perform thread swaps (migrations) between different core types every so often. Because frequent thread migrations may introduce significant overheads, the scheduler does not trigger a thread swap as soon as it detects that a thread T_B running on a big core has a greater `amp_vruntime` than that of a thread T_S running on a small core. Instead, the ACFS scheduler swaps T_B and T_S in the event ($\texttt{amp\_vruntime}_{T_B} - \texttt{amp\_vruntime}_{T_S}$) $\geq$ `amp_threshold`. Increasing the `amp_threshold` makes it possible to effectively reduce the thread-swap frequency thus mitigating migration overheads. However, very high threshold values may lead to accumulating unfairness for longer periods. To cater to this tradeoff, we performed a sensitivity study and found that a value of `vruntime_threshold` that enforces an average swap rate of 850ms ensures an acceptable trade-off in our experimental setting. To achieve a target average swap rate T_{swap}, more suitable for other experimental settings, the associated threshold can be approximated as follows:

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

where SF_{avg} denotes the average speedup factor observed in the platform, and T_{tick} represents the *tick* length.

Despite ACFS's design has been described in terms of big and small cores, it could be augmented to work on AMPs with more types of cores. To this end, the definition of the slowdown factor (S_{core}) must be extended to n core types.

[3]EQP relies on either IPC sampling or PIE to estimate SFs. Since PIE is not available on existing asymmetric hardware, in this work we evaluated the IPC-sampling variant of EQP.

[4]Note than when a thread enters the system its `amp_vruntime` 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.

Table 1: Multi-application workloads

Workload	Benchmarks
4STH	calculix, gamess, GemsFDTD, bzip2
3STH-1STL	gamess, GemsFDTD, bzip2, sjeng
2STH-2STM	gamess, soplex, povray, h264ref
2STH-2STL	mcf, calculix, sjeng, gobmk
1ST{H,M}-2STL	mcf, h264ref, sjeng, gobmk
3STH-1HPH	hmmer, gobmk, h264ref, fma3d_m(9)
1ST{H,M,L}-1PSH	gamess, astar, soplex, blackscholes(9)
2PSH-1HPM	BLAST(4), semphy(4), wupwise_m(4)
1PSH-1PSL	semphy(6), FFTW3D(6)
1PSH-1HPH	BLAST(6), fma3d_m(6)

3.1 Throughput-fairness trade-off

Using the theoretical model presented in [12] and assuming perfect speedup estimations, we found that the *base* ACFS design described thus far behaves just like the Opt-Unfairness theoretical scheduler (see Section 2.1) for multi-application workloads consisting of single-threaded programs. In other words, it provides the maximum throughput attainable for the optimal unfairness in this scenario.

To make it possible for the system administrator to trade fairness for throughput, we augmented ACFS with the `unfairness_factor` (UF) knob. When this 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 greater than the default setting, the ACFS scheduler increases throughput at the expense of degrading fairness up to a certain extent. Intuitively, making this possible boils 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, which entails re-defining $\Delta_{\texttt{amp\_vruntime}}$ as follows:

$$\Delta_{\texttt{amp\_vruntime}} = \frac{100 \cdot W_{\text{def}}}{S_{core} \cdot W_t \cdot \left(1 + \frac{(UF-1)\cdot(S_{BS}-S_{min})}{S_{max}-S_{min}}\right)} \quad (5)$$

where S_{max} and S_{min} are the maximum and minimum speedups (S_{BS}s) observed among applications in the workload, respectively. Intuitively, with this new definition of $\Delta_{\texttt{amp\_vruntime}}$, the `amp_vruntime` of high-speedup threads increases at a slower pace than that of low-speedup threads, which results in a higher big-core share for high-speedup applications and, in turn, in higher system throughput.

Using the analytical model presented in [12], we observed that gradually increasing the `unfairness_factor` (UF) for a workload leads to throughput gains while ensuring unfairness no greater than $UF \cdot opt$, where *opt* denotes the optimal unfairness for the workload in question. Notably, this ideal unfairness value can be only reached with perfect speedup estimates. In Section 4.C, we analyze the effect of varying the UF in more realistic scenarios.

4. EXPERIMENTAL EVALUATION

We carried out an extensive comparison of ACFS with several state-of-the-art schedulers for AMPs: HSP [6, 11], Prop-SP [12], EQP [14], RR [3] and A-DWRR [8]. We implemented all scheduling algorithms in the Linux kernel 3.2.

For the evaluation we used the Intel QuickIA prototype system [4]. This platform consists of a dual socket UMA system featuring two multicore processors: a quad-core Intel Xeon E5450 processor, where cores operate at 1.2Ghz; and a dual-core Intel Atom N330 processor, where cores run at 1.6Ghz. To reduce shared-resource contention effects for the experiments we disabled one core on each die in the Xeon processor. This setting gives us a paring of two high-performance *big* cores (E5450) with two low power *small* ones (N330). We will refer to this asymmetric configuration as 2B-2S. Observed speedup factors for single-threaded programs on this platform range from 1.0 to 4.7.

To run workloads including multithreaded applications we opted to use an AMP configuration with a greater core count than that of 2B-2S. To this end, we also experimented with a NUMA multicore server that integrates two AMD Opteron 2425 hex-core processors. On this platform we emulated an AMP system consisting on 2 big cores and 10 small ones (2B-10S) by reducing the processor frequency on some cores; specifically, "big" cores on 2B-10S operate at 2.1GHz whereas "small" cores run at 800MHz. On this configuration we observe a range of speedup factors between 1.74 and 2.62.

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 FFT. In all experiments, the total thread count in the workload was set to match the number of cores in the platform, since this is how runtime systems typically set the number of threads for CPU-bound workloads like the ones we used. 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.

A) Applications with the same priority. We begin by analyzing the effectiveness of the various schedulers for multi-application workloads consisting of sequential and parallel programs 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 selected program mixes, shown in Table 1, mimic scenarios with different SF ranges and varying degree of competition for the scarce big cores in the AMP. Note that the workload name encodes the category of each application, as observed on the AMP system where we ran the workload. (The first five workloads are evaluated on the 2B-2S system; the remaining ones were run on 2B-10S.) For multithreaded applications, the number in parentheses by each program's name is the number of threads it runs with.

Figure 1 shows the results for the first five workloads, consisting of single-threaded applications only running on 2B-2S. Overall, 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 evident, fair-sharing big cores may lead to throughput and fairness degradation, especially for workloads exhibit-

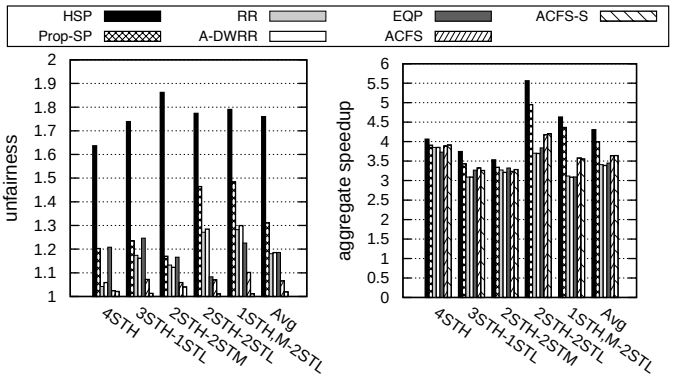


Figure 1: Fairness and throughput on 2B-2S (Intel QuickIA)

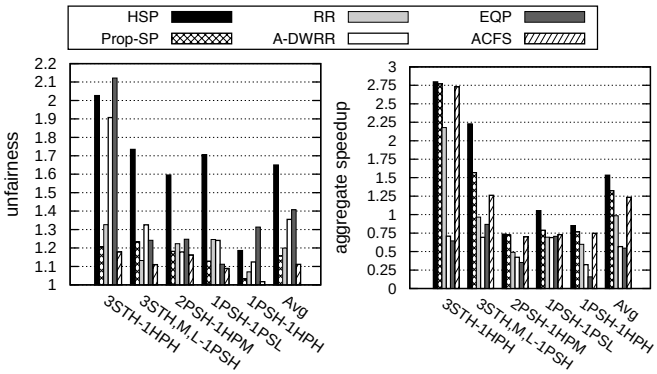


Figure 2: Fairness and throughput on 2B-10S (AMD platform)

ing a wide range of big-small speedups (e.g., 3SH-1STL and 2STH-2ST). 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 subjected to high unfairness in some cases (e.g., 2STH-2STL).

Now we zoom in on the results for the EQP and ACFS schedulers, which strive to optimize fairness. As evident, 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 keep track of threads' slowdowns, as pointed out in Section 2.3. Moreover, EQP is subjected 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 [13]. Evaluating a real-world implementation of EQP enabled us to observe this issue. (Note that in the original work [14], EQP was simulated.) The ACFS scheduler, on the contrary, predicts a thread's SF by means of an estimation model that uses performance metrics collected on the *current core type* via hardware counters, so it is not subject to the aforementioned program-phase issues. The SF estimation model we derived for the Intel QuickIA provides ACFS with SF estimates with correlation coefficients of 0.94 and 0.93 for SPEC CPU benchmarks on the big and the small core respectively. The results reveal that, despite the existing inaccuracies in the SF model, ACFS is able to obtain the best unfairness figures across the board. In particular, it reduces unfairness by 11% on average compared to EQP, RR and A-DWRR, while obtaining a 5% average increase in throughput. To evaluate the impact of SF inaccuracies on ACFS, 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. As evident, perfect overall SF values make it possible to reduce unfairness even further. This fact underscores that high accuracy in speedup estimation is paramount when it comes to delivering fairness on AMPs.

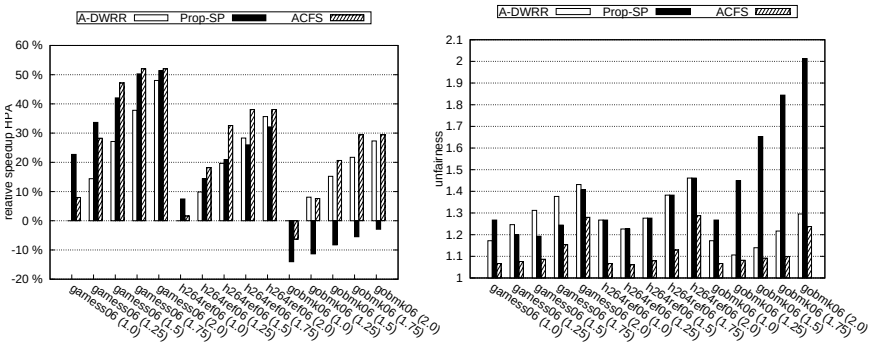
We now turn our attention to the results for workloads that include parallel and sequential applications running on 2B-10S (Figure 2). For HSP, PropSP and ACFS, we observed similar trends than those of workloads on 2B-2S: ACFS achieves the best fairness figures across the board while HSP and Prop-SP obtain slightly better throughput at the expense of degrading fairness. In this scenario, the three schedulers make scheduling decisions by taking into account the speedup the application as a whole derives from using big cores available on the AMP. EQP, on the other hand, aims to achieve equal slowdown across threads by considering the SF of individual threads only. However, ensuring that each thread in the system experiences a similar slowdown does not ensure *equal slowdowns among applications* when multithreaded programs are included in the workload. Failing to consider the application-wide speedup leads EQP to higher fairness degradation than ACFS.

Finally we look at the results of A-DWRR and RR on 2B-10S. A-DWRR ensures that each thread in the workload receives the same AMP-scaled CPU time, regardless the application it belongs to; as such, programs with a high thread count receive a high big-core share. RR, by contrast, fair-shares big cores among applications. Because applications with a high thread count usually derive low speedup from the scarce big cores [10, 1], A-DWRR is subject to higher throughput and fairness degradation than RR.

B) 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 in the 2B-2S configuration. Specifically, we experimented with a workload featuring two high-speedup applications (`gamess` and `bzip2`), one mid-speedup program (`h264ref`) and a low-speedup one (`gobmk`). Such a diverse workload enables us to explore how different applications can be accelerated as the priority increases and how unfairness is affected.

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%. Figure 3 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, we report unfairness and the HPA's relative speedup. Since we observed very similar trends for the two single-threaded high-speedup programs, we omitted the results for `bzip2`. Note 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. Notably, to factor in application priorities in the unfairness metric (as in [5]), we replaced slowdowns in Eq. 2 with their weighted counterparts: $W_t \cdot Slowdown_{app}$.

As evident, all schedulers are able to reduce the comple-



(a) HPA speedup (b) Unfairness

Figure 3: Results for applications with different priorities.

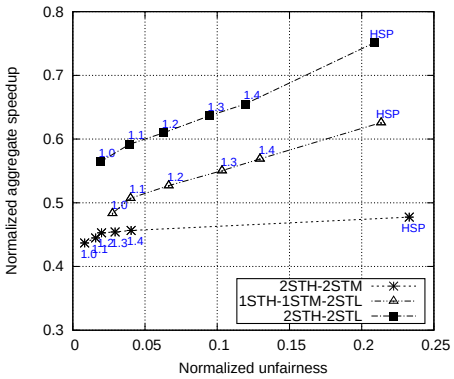


Figure 4: Unfairness vs. throughput for different UFs.

tion time of the HPA as the priority increases. This stems from the fact that they all gradually increase the HPA's big-core share with the priority. Notably, the results reveal that only ACFS is able to maintain low unfairness as the HPA's priority increases. Prop-SP and A-DWRR, by contrast, are subjected to high fairness degradation. Throughput results, omitted due to space constraints, reflect similar trends to those observed in scenarios with equal-priority applications.

C) Trading fairness for throughput. 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. Figure 4 shows how the UF choice affects fairness and throughput for three selected workloads (Table 1). 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. 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. CONCLUSIONS

In this paper we proposed ACFS, a scheduler that seeks to optimize fairness on AMPs while maintaining acceptable system throughput. To this end, ACFS evens out the slowdown that the various applications in a multi-program workload experience as a result of sharing the asymmetric system. To track the slowdown accumulated by an application 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 11% on average compared to the RR, A-DWRR and EQP schemes, while providing higher system throughput. We also demonstrated that previous schedulers that support user priorities on AMPs, such as Prop-SP, are subject to high fairness degradation in the event applications with different priorities coexist on the system; ACFS, by contrast, ensures low unfairness in this scenario. Key elements for the success of ACFS are the mechanism used to keep track of the slowdown accumulated by the applications and its reliance on online estimation models to approximate speedup factors.

Acknowledgements

This work has been supported by the Spanish government through the research contract TIN2012-32180. We would like to thank David Koufaty (Circuits and Systems Research Lab at Intel) and Alexandra Fedorova for enabling us to experiment with the Quick-IA prototype system.

6. REFERENCES

[1] M. Annavaram et al. Mitigating Amdahl's Law through EPI Throttling. In *Proc. of ISCA'05*, 2005.

[2] ARM. Benefits of the big.LITTLE Architecture. 2012.

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

[4] N. Chitlur et al. QuickIA: Exploring heterogeneous architectures on real prototypes. In *Proc. of HPCA'12*.

[5] E. Ebrahimi et al. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. ASPLOS '10, 2010.

[6] D. Koufaty et al. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proc. of Eurosys '10*.

[7] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. of ISCA '04*, 2004.

[8] T. Li et al. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proc. of HPCA'10*, 2010.

[9] M. Pricopi et al. Power-performance modeling on asymmetric multi-cores. In *Proc. of CASES '13*, 2013.

[10] J. C. Saez et al. A Comprehensive Scheduler for Asymmetric Multicore Systems. In *Proc. of Eurosys '10*, 2010.

[11] J. C. Saez et al. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM TOCS*, 30(2), Apr. 2012.

[12] J. C. Saez et al. Exploring the throughput-fairness trade-off on asymmetric multicore systems. In *Proc. of ROME '14*, 2014.

[13] D. Shelepov et al. HASS: a Scheduler for Heterogeneous Multicore Systems. *ACM OSR*, 2009.

[14] K. Van Craeynest et al. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *PACT'13*.

[15] K. Van Craeynest et al. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proc. of ISCA '12*, 2012.