© The British Computer Society 2017. All rights reserved.
For Permissions, please e-mail: journals.permissions@oup.com
doi:10.1093/comjnl/bxx038

On the Interplay Between Throughput, Fairness and Energy Efficiency on Asymmetric Multicore Processors

J. C. SAEZ[1*], A. POUSA[2], A. E. DE GIUSTI[2] AND M. PRIETO-MATIAS[1]

[1]*ArTeCS Group, Facultad de Informática, Complutense University of Madrid, Madrid, Spain*
[2]*III-LIDI, Facultad de Informática, National University of La Plata, La Plata, Argentina*
Corresponding author: jcsaezal@ucm.es

Asymmetric single-ISA multicore processors (AMPs), which integrate high-performance big cores and low-power small cores, were shown to deliver higher performance per watt than symmetric multicores. Previous work has highlighted that this potential of AMP systems can be realizable by scheduling the various applications in a workload on the most appropriate core type. A number of scheduling schemes have been proposed to accomplish different goals, such as system throughput optimization, enforcing fairness or reducing energy consumption. While the interrelationship between throughput and fairness on AMPs has been comprehensively studied, the impact that optimizing energy efficiency has on the other two aspects is still unclear. To fill this gap, we carry out a comprehensive analytical and experimental study that illustrates the interplay between throughput, fairness and energy efficiency on AMPs. Our analytical study allowed us to define the *energy-efficiency factor* (EEF) metric, which aids the OS scheduler in identifying which applications are more suitable for running on the various cores to ensure a good balance between performance and energy consumption. We propose two energy-aware OS-level schedulers that leverage the EEF metric; the first one strives to optimize the energy-delay product and the second scheduler can be configured to optimize different metrics on the AMP. To demonstrate the effectiveness of these proposals, we performed an extensive evaluation and comparison with state-of-the-art schemes by using real asymmetric hardware and scheduler implementations in the Linux kernel.

Keywords: asymmetric multicore; scheduling; operating systems; energy efficiency; fairness

Received 11 July 2016; revised 31 January 2017; editorial decision 23 March 2017
Handling editor: Iain Stewart

1. INTRODUCTION

Previous research has highlighted that asymmetric single-ISA (instruction set architecture) multicore processors (AMPs), which couple same-ISA complex high-performance big cores with power-efficient small cores on the same chip, have been shown to significantly improve upon the energy and power efficiency of their symmetric counterparts [1]. The ARM big. LITTLE processor [2] and the Intel QuickIA prototype [3] demonstrate that AMP designs have drawn the attention of major hardware players.

Despite their benefits, AMPs pose significant challenges to the OS scheduler [4]. One of the main challenges is how to effectively distribute big-core cycles among the various applications running on the system. Most existing scheduling schemes have focused on maximizing the system throughput for multi-application workloads [1, 5–8]. 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 [1]. Further throughput gains can be achieved by using big cores to accelerate sequential phases of parallel programs [6, 9, 10].

Notably, asymmetry-aware schedulers that maximize throughput alone are known to be inherently unfair [11]. Unfairness typically gives rise to undesirable effects on the system [12, 13]. For example, an application may experience very different completion times from run to run, depending on the co-running applications [14]. Moreover, equal-priority applications may not experience the same performance penalty (slowdown) when running together relative to their performance when running alone on the AMP. This makes priority-based

scheduling policies ineffective, reduces performance predictability and can lead to wrong billings in commercial cloud-like computing services, where users are charged for CPU hours. These QoS-related issues can be addressed on AMPs via fairness-aware scheduling algorithms [11, 14–16].

Previous research has also highlighted that maximizing system throughput on AMPs does not always result in a good performance-energy consumption trade-off [17]. Specifically, using big cores to run applications in the workload that derive the highest big-to-small relative speedup does not always constitute the most energy-efficient schedule. In the quest of higher energy efficiency on AMPs, several scheduling schemes have been recently proposed [17, 18].

While previous work has shown that the goals of system throughput optimization and fairness optimization on AMPs are largely conflicting scheduling objectives [11, 14], no one has yet analyzed the interrelationship between energy efficiency, throughput and fairness. In this paper, we carry out a comprehensive analytical and experimental study that illustrates how optimizing one of these aspects alone affects the other two. Specifically, our paper makes the following main contributions:

- We derived an analytical model to approximate the system throughput, degree of fairness and energy efficiency that a given scheduler delivers for multi-program workloads on AMPs. By using a simulator that relies on this model, we find the schedule that optimizes each metric for different synthetic workloads. Our study reveals that optimizing the energy-delay product (EDP)—a proxy for assessing the degree of energy efficiency—may come at the expense of serious fairness and throughput degradation. This theoretical study also allowed us to define the *energy-efficiency factor* (EEF) metric, a key heuristic enabling the scheduler to substantially reduce the EDP for multi-program workloads on an asymmetric multicore system.
- We propose two novel asymmetry-aware OS-level scheduling algorithms that leverage the EEF metric. The first scheduler, referred to as EEF-Driven, approximates the theoretical scheduler that delivers the maximum throughput attainable for the optimal (minimal) EDP value. The second scheduler, named ACFS-E, is a variant of the fairness-aware ACFS scheduler [14]. ACFS-E is equipped with configurable parameters enabling the system administrator to gradually trade fairness for energy efficiency or throughput. Moreover, to the best of our knowledge, ACFS-E constitutes the first asymmetry-aware scheduling scheme that can be configured to optimize fairness, energy efficiency or throughput, by employing a single yet flexible algorithm.
- We implemented the EEF-Driven and ACFS-E schedulers in the Linux kernel. Specifically, the implementation of these algorithms relies on the ability of the OS to determine the EEF of a thread at runtime. To determine this factor online on existing asymmetric hardware, we built performance-counter-based prediction models by leveraging a variant of the methodology proposed in a previous work [19], which was originally devised to aid in predicting a thread's cross-core relative performance. In relying on prediction models, the implementation of EEF-Driven and ACFS-E does not require changes in the applications or special hardware extensions.
- To assess the effectiveness of the proposed OS-level scheduling schemes, we employed a commercial asymmetric multicore platform that features an ARM big.LITTLE processor [20]. On this platform, we performed an extensive experimental comparison with state-of-the-art asymmetry-aware schemes [5, 6, 14, 17]. Notably, some of these previously proposed schemes were evaluated before on emulated asymmetric hardware [5, 6] or simulators [17]. Instead, we implemented these schemes in the Linux kernel and performed an extensive evaluation on real asymmetric hardware. Our experimental analysis (Section 5) corroborate the conclusions drawn from our theoretical study (Section 2). Moreover, the results reveal that EEF-Driven improves throughput (by up to 20%) and reduces the EDP (by up to 15%) compared to a state-of-the-art energy-aware scheme [17]. Furthermore, our study demonstrates that ACFS-E makes a versatile scheduling scheme, as it enables the administrator to (i) optimize different metrics in the AMP with a single algorithm, and (ii) to trade fairness for energy efficiency or throughput in scenarios where fairness constraints are relaxed.

The rest of the article is organized as follows. Section 2 presents the theoretical model proposed in this work and showcases the results of our analytical study. Section 3 describes related work. Section 4 outlines the design of the EEF-Driven and ACFS-E schedulers. Section 5 presents the results of our experimental evaluation, and Section 6 concludes.

2. ANALYTICAL STUDY: SYSTEM THROUGHPUT, FAIRNESS AND ENERGY EFFICIENCY ON AMPS

In this section, we begin by introducing the metrics used to quantify throughput, energy efficiency and fairness on AMPs. Then, we present the synthetic scenario considered for our study, the theoretical model derived and the scheduling algorithms considered. Finally, we proceed to discuss the results of our theoretical study.

2.1. Metrics

Previous research on fairness for CMPs [12, 13, 21] and AMPs [11, 16] define a scheme as fair if equal-priority applications in a multi-program workload suffer the same

slowdown due to sharing the system. To cope with this notion of fairness, we employed the (lower-is-better) *unfairness* metric [13], which is defined as follows:

$$\text{Unfairness} = \frac{\text{MAX}(\text{Slowdown}_1, \ldots, \text{Slowdown}_n)}{\text{MIN}(\text{Slowdown}_1, \ldots, \text{Slowdown}_n)}, \quad (1)$$

where $\text{Slowdown}_i = \frac{\text{CT}_{\text{sched},i}}{\text{CT}_{\text{fast},i}}$. In turn, $\text{CT}_{\text{sched},i}$ denotes the completion time of application i under a given scheduler, and $\text{CT}_{\text{fast},i}$ is the completion time of application i when running alone on the AMP (with all the big cores available to the application).

To quantify throughput on AMPs, previous work [11, 14] has employed the *Aggregate SPeedup* (ASP) metric, defined as follows:

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

where n is the number of applications in the workload, and $\text{CT}_{\text{slow},i}$ is the completion time of application i when it runs alone on the AMP and uses small cores only. The ASP metric (higher-is-better) captures the overall efficiency that a workload derives from the various cores under a particular scheduler.

To assess the energy efficiency of a workload under a particular scheduler, we use the (lower-is-better) EDP [22–24] metric:

$$\begin{aligned}\text{EDP} &= \frac{\text{Total_Energy_Consumed}}{\text{Instructions_per_second}} \\ &= \frac{\text{Total_Energy_Consumed} * T}{\text{Total_retired_instructions}},\end{aligned} \quad (3)$$

where T is the workload's completion time. Equation (3) was extracted from [24].

2.2. Synthetic scenario

To illustrate the interrelationship between the aforementioned three metrics, we carry out a theoretical study on the effectiveness of different scheduling algorithms when running several synthetic multi-program workloads on an AMP system consisting of two big cores and two small cores. All workloads comprise four compute-intensive single-threaded applications each. In this hypothetical scenario, we assume that applications exhibit constant big-to-small performance ratios and uniform energy-per-instruction (EPI) ratios on each core type throughout the execution. For single-threaded applications, the big-to-small speedup matches the *speedup factor* (SF) of its single runnable thread, defined as $\frac{\text{IPS}_{\text{big}}}{\text{IPS}_{\text{small}}}$, where IPS_{big} and $\text{IPS}_{\text{small}}$ are the thread's instructions per second (IPS) ratios achieved on big and small cores, respectively.

In building the synthetic workloads, we analyzed the behavior of various benchmarks from SPEC CPU running on the ARM Juno Development board [20]. This system features a 64 bit ARM big.LITTLE processor consisting of Cortex A53 *small* cores and Cortex A57 *big* cores (More information on this platform can be found in Section 5). We used the performance counters and energy registers integrated in the ARM Juno board to obtain the IPC and EPI values for different program phases on the various core types for the benchmarks. For each benchmark, we picked a program phase exhibiting stable IPC and EPI ratios on both core types for several contiguous samples.[1] Each synthetic application used in our study represents one of these stable program phases.

Table 1 shows the properties of the various synthetic applications considered. For each application, the associated SPEC CPU benchmark, the SF, as well as the IPC and the EPI values (specified in nanojoules/instr.) for both core types are displayed. In measuring EPI values, we factored in both the energy consumption of the corresponding CPU core cluster (big or little) and the net DRAM energy consumption while the benchmark runs alone on the system. Note that OS scheduling decisions (e.g. thread-to-core mappings) can largely affect the energy consumption of CPU cores and DRAM when running workloads consisting of CPU-intensive and memory-intensive applications, such as those considered in our analysis. As such, in this work we opted to factor in the energy consumption of both platform components.

2.3. Theoretical model

In an earlier work [11], we proposed a set of analytically derived formulas to approximate the ASP and unfairness of a workload under a particular scheduler in the synthetic scenario described in the previous section. These formulas make it possible for us to find the schedule (i.e. big-core cycle distribution across applications) that optimizes throughput, fairness or guarantees a specific trade-off between both metrics [11]. Our main goal is to augment that theoretical model with a mechanism to approximate the EDP metric in the synthetic scenario considered. This is crucial to find the schedule that optimizes the EDP for diverse workloads.

Before proceeding with the derivation of the formulas enabling to approximate the EDP, we present the basics of the theoretical model proposed in [11], which consists of the following equations:

$$\text{ASP} = \sum_{i=1}^{n} \left(\frac{1}{\frac{f_i}{SF_i} + (1 - f_i)} - 1 \right) \quad (4)$$

[1] We sampled performance counters and energy registers every 500 million instructions. For this task, we turned to the *Event-Based Sampling* feature of the PMCTrack tool [25].

TABLE 1. Synthetic applications.

Applications	Benchmark	IPC_{big}	IPC_{small}	SF	EPI_{big} (nJ/instr)	EPI_{small} (nJ/instr)
A1	art	0.60	0.24	2.47	1.59	1.86
A2	astar	0.58	0.31	1.86	1.40	1.10
A3	bzip2	1.49	0.73	2.02	0.61	0.45
A4	equake	0.80	0.26	3.07	1.31	1.45
A5	galgel	1.30	0.41	3.16	0.82	0.91
A6	gamess	2.01	0.69	2.91	0.51	0.49
A7	gobmk	1.09	0.61	1.79	0.75	0.54
A8	gzip	1.07	0.63	1.70	0.78	0.56
A9	h264ref	1.83	0.93	1.96	0.51	0.37
A10	hmmer	2.80	1.04	2.69	0.42	0.36
A11	mcf	0.18	0.09	2.02	3.98	4.44
A12	mgrid	1.28	0.59	2.17	0.85	0.65
A13	perlbench	1.42	0.71	2.01	0.60	0.47
A14	perlbmk	1.77	0.78	2.27	0.54	0.41
A15	povray	1.15	0.53	2.19	0.85	0.66
A16	soplex	0.52	0.20	2.53	1.68	1.86
A17	swim	0.25	0.11	2.24	3.11	3.34
A18	vortex	1.73	0.72	2.41	0.56	0.45
A19	wupwise	1.63	0.64	2.56	0.66	0.54

$$\mathrm{Slowdown}_i = f_i + SF_i \cdot (1 - f_i) \tag{5}$$

$$f_i = \frac{1}{\frac{1}{SF_i} \cdot \left(\frac{1}{F_i} - 1\right) + 1}, \tag{6}$$

where F_i denotes the big-core time fraction allotted by a given scheduler to application i throughout the execution, f_i represents the fraction of instructions over the total that application i completes on a big core under the scheduler in question, and SF_i is the application's SF. Despite the fact that F_i and f_i are related (as shown in Equation (6)), obtaining F_i for an application under a particular scheduler is more straightforward than obtaining f_i; we elaborate on this aspect in Section 2.4. Hence, Equation (6) is provided for the sake of convenience. Overall, Equations (1) and (4)–(6) make it possible to approximate the ASP and unfairness of a workload under a particular scheduler.

To make the derivation of the aforementioned formulas tractable, some simplifying assumptions were made in [11]. First, the number of applications (n) does not exceed the total number of cores in the AMP: $N_{BC} + N_{SC} \geq n$, where N_{BC} and N_{SC}, denote the number of big and small cores in the AMP, respectively. Second, all applications in the workload run continuously on the system for a certain amount of time T. The analytical unfairness and throughput (ASP) values approximate the actual values corresponding to that time interval. Third, the scheduler is work conserving and keeps big cores busy to improve throughput. Therefore, an application's small-core fraction is $1 - F_i$, and we have that $0 \leq F_i \leq 1$ and $\sum_{j=1}^{n} F_j = N_{BC}$. Fourth, the model does not factor in overheads due to shared-resource contention or thread migrations. Note that we do account for these overheads in our experimental evaluation (Section 5).

We now proceed to derive a set of analytical formulas making it possible to approximate the EDP for synthetic workload scenarios. Let $EC_{sched,i}$ and $EPI_{sched,i}$ be the energy consumption and EPI ratio, respectively, for application i when running for T seconds under a particular scheduler. Let $IPS_{sched,i}$ be the number of IPS achieved by application i. Suppose further that NI_i denotes the total number of instructions that application i completes in T seconds under the scheduler in question. Hence, we can derive the analytical EDP as follows:

$$\begin{aligned} \mathrm{EDP} &= \frac{\mathrm{Total_Energy_Consumed}}{\mathrm{Instructions_per_second}} \\ &= \frac{\sum_{i=1}^{n} EC_{sched,i}}{\sum_{i=1}^{n} IPS_{sched,i}} = \frac{\sum_{i=1}^{n} NI_i \cdot EPI_{sched,i}}{\sum_{i=1}^{n} IPS_{sched,i}} \\ &= \frac{\sum_{i=1}^{n} T \cdot IPS_{sched,i} \cdot EPI_{sched,i}}{\sum_{i=1}^{n} IPS_{sched,i}} \\ &= T \cdot \frac{\sum_{i=1}^{n} IPS_{sched,i} \cdot EPI_{sched,i}}{\sum_{i=1}^{n} IPS_{sched,i}} \end{aligned} \tag{7}$$

In turn, $\text{IPS}_{\text{sched},i}$ [11] and $\text{EPI}_{\text{sched},i}$ can be defined in terms of f_i as well as the various application properties shown in Table 1, as follows:

$$\text{IPS}_{\text{sched},i} = \frac{\text{IPS}_{\text{big},i}}{f_i + \text{SF}_i \cdot (1 - f_i)} \tag{8}$$

$$\text{EPI}_{\text{sched},i} = \text{EPI}_{\text{big},i} \cdot f_i + \text{EPI}_{\text{small},i} \cdot (1 - f_i) \tag{9}$$

Note that $\text{EPI}_{\text{big},i}$ and $\text{EPI}_{\text{small},i}$ represent the EPI rates for application i when it runs on a big and on a small core, respectively.

2.4. Scheduling algorithms

For our analytical study, we consider four asymmetry-aware schedulers. The first one, denoted as High-SPeedup (*HSP*) [1, 5, 6, 8], optimizes throughput by using big cores to run 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 (i.e. $F_i = 0$). The second scheduler, referred to as *Opt-Unf*, constitutes a theoretical schedule that ensures the maximum ASP (throughput) value attainable for the optimal unfairness. (As shown in [19], the ACFS scheduler makes it possible to approximate the behavior of *Opt-Unf*.) The third scheduler considered, denoted as *Opt-EDP*, is also a theoretical scheme; it ensures the maximum ASP value attainable for the optimal EDP. Finally, the fourth scheduling algorithm is referred to as *EEF-Driven*, one of the schedulers proposed in this work. Essentially, EEF-Driven maps to big cores the N_{BC} applications that yield the highest *EEF*; the remaining applications are mapped to small cores. An application's EEF is defined as follows: $\frac{\text{SF}}{\text{EPI}_{\text{big}}}$, where EPI_{big} is given in nanojoules/instr. Notably, we arrived at this factor by analyzing the thread-to-core mappings performed by the Opt-EDP scheduler. Specifically, EEF-Driven strives to approximate the behavior of this theoretical schedule, by using the EEF of the various applications, which can be estimated at run time on commercial AMPs (as shown in Section 4.2).

While determining the big-core cycle distribution among applications (F_i) for the HSP and EEF-Driven is straightforward in this synthetic scenario—as the application properties shown in Table 1 are known—obtaining this information under the theoretical approaches (Opt-Unf and Opt-EDP) requires an extensive exploration of the search space. To determine the big-core cycle distribution among applications in this context, we created a simulator that makes use of Equations (1) and (4)–(9), and finds the optimal solution in each case via a branch-and-bound algorithm. For example, for the Opt-Unf scheduler, 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 $\sum F_i = N_{\text{BC}}$. In addition, simple heuristics are used to prune unpromising solutions. A similar approach is used to find the optimal schedule for Opt-EDP. In that case, however, the IPS, EPI and SF for each application in the workload are factored in, to be able to determine the EDP associated with the various solutions explored. Note also that in determining optimal schedules, we assume that the frequency of each core cluster (big or small) remains at the default setting throughout the execution. While lowering the processor frequency may help reducing energy consumption on AMPs [26], it may also degrade application performance as well, which has a negative impact in throughput and fairness. Analyzing the effects of OS-level asymmetry-aware scheduling coupled with DVFS policies is out of the scope of this paper, but constitutes an interesting avenue for future work.

2.5. Discussion

To evaluate the effectiveness of the aforementioned algorithms regarding throughput, fairness and energy efficiency, we built different workloads consisting of combinations of the synthetic applications from Table 1. In creating the workloads, we filtered out those application mixes where the HSP and the Opt-EDP schedulers perform the same big-core cycle distribution, as these workloads yield the same ASP, unfairness and EDP values. Table 2 shows a representative subset of the workloads we explored. The results associated with these workloads are displayed in Fig. 1a and b; the first figure shows an ASP-versus-EDP graph, whereas the second one displays unfairness versus EDP.

The results reveal that the three metrics cannot be optimized simultaneously. Clearly, HSP achieves the best throughput figures across the board, while achieving EDP values considerably higher (worse) than the optimal (up to a

TABLE 2. Workloads.

Workload	Applications
W1	A5, A4, A6, A10
W2	A16, A17, A13, A9
W3	A16, A1, A18, A14
W4	A5, A4, A10, A15
W5	A5, A4, A6, A12
W6	A1, A12, A3, A13
W7	A1, A17, A7, A8
W8	A15, A11, A13, A2
W9	A4, A11, A3, A8
W10	A10, A19, A16, A9

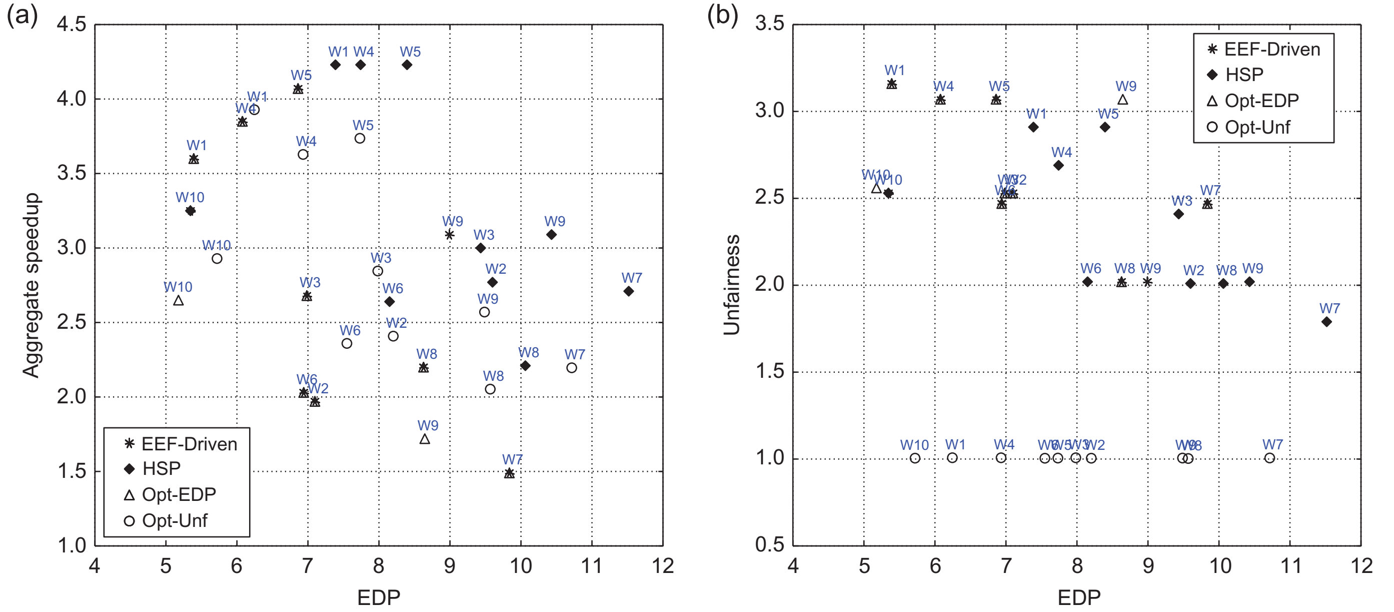

FIGURE 1. ASP, EDP and unfairness values for workloads from Table 2 under the various schedulers. In the left figure, the closer to the top left corner, the better the result. In the figure on the right, better results are concentrated towards the bottom left corner. Note that we used $T = 10$ s to obtain EDP estimations with the simulator. Nevertheless, the trends remain the same regardless of the value of T. (**a**) EDP vs. ASP, (**b**) EDP vs. Unfairness.

22% increase). Conversely, Opt-EDP yields the minimum EDP value for all workloads, but this comes at the expense of crude throughput degradation in some cases (up to 44% for W9). Note also that Opt-EDP and HSP are inherently unfair; both schedulers reap throughput benefits and energy savings at the cost of significant fairness degradation (values considerably higher than 1). As for EEF-Driven, we observed that it enforces the same big-core cycle distribution than Opt-EDP in most cases, which leads to identical EDP, throughput and fairness results. More importantly, in those workload scenarios where EEF-Driven and Opt-EDP perform a different big-core cycle distribution, such as W10 and W9, EEF-Driven yields a very small increase in EDP (up to 4%) while improving throughput significantly. Hence, the theoretical results suggest that EFF-Driven constitutes a good approximation of Opt-EDP.

The workloads explored thus far consist of synthetic applications that represent program phases of SPEC CPU benchmarks running on a 64 bit big.LITTLE processor. To verify that the conclusions drawn so far are also valid on a different AMP platform, we also experimented with the Odroid XU4 board [27]. This system features a 32 bit big.LITTLE processor that integrates a mix of Cortex A7 *small* cores, and Cortex A15 *big* cores. On such a system, the SF and EPI ranges for SPEC CPU benchmarks are very different to those observed on the big.LITTLE processor of the Juno board. This stems from the fact that cores of the same category (i.e. big or small) on these platforms differ in microarchitectural features, processor frequency and power consumption [20, 27]. Specifically, we observed that the power consumption of a big core on the 32 bit processor (Odroid XU4) is roughly four times the power consumption of a big core on the 64 bit processor (Juno). Under these circumstances, to cover a wide spectrum of performance and energy consumption profiles for the Odroid board, we had to select a specific set of representative program phases. To that end, we ran different SPEC CPU benchmarks alone on big and small cores, and identified program phases with a stable value of IPS and EPI. Note that, unlike the Juno board, the Odroid XU4 does not feature built-in energy registers. To measure energy consumption on this system, we turned to an external power monitor (the Odroid Smart Power), from which we obtain data via the PMCTrack monitoring tool [25].

By using the phase-level information gathered for various SPEC CPU benchmarks on the Odroid XU4 board, we defined a set of synthetic applications, whose properties are shown in Table 3. We then built different 4-application mixes using these synthetic applications, by following the same procedure used for the Juno board. Table 4 shows a representative subset of the workloads we explored for a hypothetical AMP system consisting of two big cores and two small cores. The results associated with these workloads are displayed in Fig. 2a and b. These results exhibit similar trends as those observed for the synthetic workloads on the 64 bit big.LITTLE processor. Again, the HSP scheduler is capable of delivering the best throughput values at the expense of

TABLE 3. Synthetic applications.

Applications	Benchmark	IPC_{big}	IPC_{small}	SF	$EPI_{big}(nJ/instr)$	$EPI_{small}(nJ/instr)$
B1	astar	0.76	0.37	2.05	4.53	1.70
B2	crafty	2.46	1.00	2.47	1.54	0.54
B3	equake	0.54	0.19	2.83	6.32	3.30
B4	gobmk	1.68	0.76	2.21	2.02	0.69
B5	h264ref	3.08	1.04	2.97	1.31	0.57
B6	mcf	0.29	0.08	3.52	10.57	7.94
B7	mgrid	2.53	0.74	3.44	1.86	0.94
B8	perlbench	2.45	1.00	2.45	1.46	0.51
B9	perlbmk	2.80	1.04	2.68	1.36	0.54
B10	povray	2.15	0.74	2.90	1.91	0.62
B11	sixtrack	1.90	0.48	3.92	1.90	1.01
B12	soplex	0.62	0.19	3.29	5.71	3.46
B13	swim	0.75	0.23	3.21	4.89	2.93

substantial EDP degradation (up to 65%). We also observe that the EDP degradation with respect to Opt-EDP is higher on this system, due to the wider range of EPI ratios observed on this platform. The results reveal that in this scenario EEF-Driven and Opt-EDP perform the same big-core cycle distribution for all the workloads, so EEF-Driven always achieves the maximum ASP value for the optimal EDP. More importantly, among every possible 4-program combination of the synthetic applications in Table 3, we did not found any workload where EEF-Driven and Opt-EDP perform differently in this scenario.

To sum up, our theoretical results highlight that fairness, throughput and energy efficiency clearly constitute conflicting objectives on AMPs, as attempting to optimize one metric may backfire by degrading the others substantially. In particular, delivering acceptable fairness (unfairness values close to one) usually comes at the expense of degrading both throughput and energy efficiency. Finally, the results also demonstrate that EEF-Driven constitutes a good approximation of the scheduler that optimizes the EDP.

TABLE 4. Workloads.

Workload	Applications
X1	B6, B12, B5, B2
X2	B12, B3, B9, B8
X3	B11, B6, B5, B3
X4	B11, B6, B5, B1
X5	B10, B3, B9, B8
X6	B7, B13, B5, B9
X7	B11, B6, B7, B5
X8	B11, B6, B5, B10
X9	B11, B3, B9, B4
X10	B6, B7, B5, B4

3. RELATED WORK

A large body of work has advocated the benefits of AMPs over symmetric CMPs [1, 28, 29]. Despite these benefits, AMP systems pose significant challenges to the system software [4]. OS scheduling is one of the most critical challenges and this is the focus of our article.

Several scheduling schemes have been proposed to improve fairness, system throughput and energy efficiency alone on AMPs. Nevertheless, no previous work has analyzed the interrelationship between these three optimization goals. Our work fills this gap and, at the same time, illustrates the capabilities of various scheduling algorithms to optimize these potentially conflicting metrics. Notably, we also propose ACFS-E, a versatile scheduling scheme that be configured to optimize any of the aforementioned three metrics with a single algorithm. The design, implementation and evaluation of this OS-level scheduler constitutes a key contribution of our research.

In the remainder of this section, we first cover scheduling proposals that seek to optimize throughput and then outline schemes designed to improve fairness. Finally, we recap previous work that focuses on reducing energy consumption on AMPs.

3.1. 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*; in other words, it must preferentially use big cores to run those applications that derive a higher benefit or *speedup* from big cores. The main difference between the available variants of the HSP approach [1, 5–8] lies in the mechanism employed to obtain threads' SFs online. Three

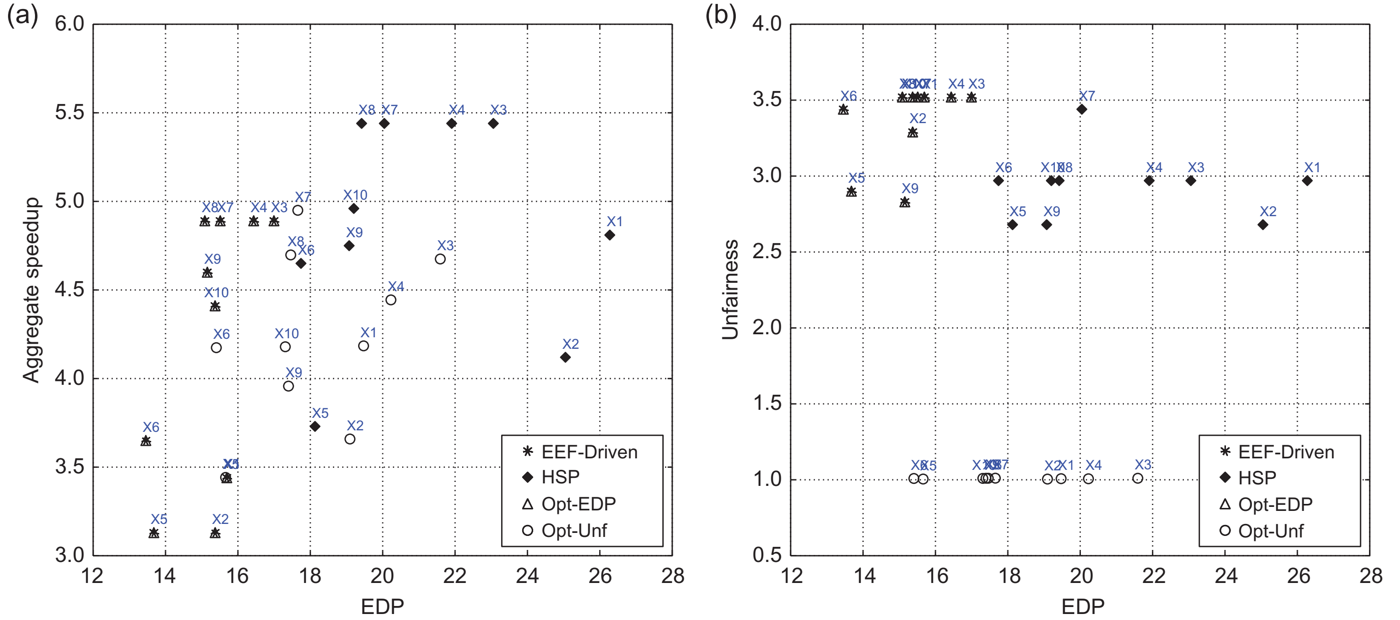

FIGURE 2. ASP, EDP and unfairness values for workloads from Table 4. (**a**) EDP vs. ASP, (**b**) EDP vs. Unfairness.

techniques have been explored to do so. The first approach comes down to measuring SFs directly [1, 8], 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 subject to inaccuracies associated with program-phase changes [7, 19]. The second approach relies on *estimating a thread's SF* using its runtime properties collected on any core type online using performance counters [5, 6, 18]. The third technique is PIE [30], a hardware-aided mechanism that has been shown to provide accurate SF estimates. Unfortunately, PIE poses certain shortcomings that renders it difficult its integration on real hardware [31].[2]

Due to the limitations of IPC sampling and PIE, we opted to use the second approach to determine a thread's SF online in the implementation of the scheduling algorithm evaluated in this work. Note that we also rely on estimation models to approximate the EEF of a thread. As discussed in Section 4.2, direct measurement of this factor is not feasible in current AMP hardware.

Other researchers proposed specific support for multi-threaded applications running on AMPs [6, 9, 32–34]. Most of these proposals target HPC applications, and make use of big cores in the AMP to accelerate serial execution phases and other scalability bottlenecks in parallel applications by employing software [6, 9] or hardware-aided approaches [32, 33]. Other authors have devised mechanisms to provide better support for irregular non-scalable applications present in desktop workloads [34]. The OS-level schedulers considered in our work are largely orthogonal to these approaches.

3.2. 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 triggering periodic thread migrations [8]. This algorithm can be easily implemented in most OSes and does not require hardware support. A hardware implementation of this scheme has also been explored [35]. Fair-sharing big cores has proven to provide more repeatable completion times across runs on AMPs [15] than default schedulers in general-purpose OSes, which are largely asymmetry agnostic, and also delivers better performance [7]. For this reason, RR has been widely used as a baseline for comparison [6, 8, 36].

Despite the fact that RR constitutes a suboptimal fairness solution [11, 19], we observed that RR does ensure repeatable completion time across runs for compute-intensive workloads, like the ones we used in our experiments. This is, however, not the case under the default scheduler of the Linux mainstream kernel (CFS), which is asymmetry unaware, and also not the case under the variant of this scheduler for AMP systems (provided by the HMP patch [37]). These two schedulers may map the same application to different core types in

[2]Despite the practical limitations of PIE, we strongly believe that adding hardware support for accurate SF estimation is a promising research avenue that could bring important benefits.

different runs of the same compute-intensive multi-program workload. On our experimental platform, these nearly random thread-to-core mappings lead to an enormous variation in the completion time of an application (up to a 2.9× increase with respect to the fastest run), which causes unrepeatable values for the ASP, the EDP and the unfairness in different runs of the same workload. Under these circumstances, and to avoid misleading conclusions from our analysis, we opted not to discuss the results of these schedulers in our experimental evaluation (Section 5). Instead, we provide the results of the RR scheme for comparison purposes.

Li *et al*. [15] 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. Note that A-DWRR, and other schemes based on it [38], do not take into account the fact that applications derive different SFs when using big cores (at a fixed processor frequency) on the platform, and that these speedups may vary over time as an application goes through different program phases. As shown in [14], this leads A-DWRR to unfairness and throughput degradation. Other researchers [11] have attempted to deliver fairness and priority enforcement on AMPs by assigning big-core share to applications in proportion to the application speedup and its priority. Nevertheless, this approach is also known to constitute a suboptimal fairness solution as it does not guarantee that equal-priority applications experience similar slowdowns (fairness) when sharing the asymmetric multicore system [14].

The ACFS scheduler [14] was designed to optimize fairness on AMPs. To this end, it leverages per-thread SF values to continuously track the relative progress that each thread in the workload makes on the AMP, and enforces fairness by evening out the slowdown observed across applications. A brief description on how the ACFS algorithm works can be found in Section 4.3. The experimental analysis presented in [19] demonstrates that ACFS clearly outperforms previous fairness-aware schemes [8, 11, 15, 16] for a wide range of workloads running on real asymmetric hardware. At the same time, unlike other asymmetry-aware schedulers that also support priorities [11], ACFS effectively maintains low unfairness in scenarios where applications with different priorities coexist on the system. Given the effectiveness of ACFS, we considered this algorithm in our experimental evaluation (Section 5.2).

More recently, Kim and Huh [39] propose different scheduling schemes, which do not try to minimize the unfairness metric, but instead strive to strictly limit the performance degradation suffered by individual applications in a workload. Notably, in their analysis, the degradation is measured with respect to the performance achieved under a scheduler that fair-shares big cores across applications (i.e. the RR scheme). This stands in contrast with measuring the performance degradation relative to running each application alone on the system (aka. slowdown), as done in previous work [13, 14, 16, 21] and as we do in this paper. We observed that reporting performance degradation with respect to a specific scheduler may mask the negative effects of shared-resource contention, which plays and important role in assessing the effectiveness of the various schemes (as we discuss in Section 5.2). Hence, we employ the widely used notion of fairness presented in Section 2.1, instead of that used in [39].

3.3. Improving energy efficiency

Other researchers have devised ways to reduce energy and power consumption on AMPs [17, 18, 40, 41]. Mogul *et al*. [40] proposed using small cores in asymmetric multicores to execute system calls. They modified an operating system to switch the execution to a less powerful core when a thread invokes a system call. The effectiveness of this scheme relies on the observation that system calls and OS code in general use big high-performance cores inefficiently. In a similar vein, Kumar and Fedorova [41] proposed binding the control domain *dom*0 of the Xen hipervisor to slow cores. These approaches are orthogonal to our proposal.

The closest proposal to our EEF-Driven scheduler is the PRIM scheme, proposed by Zhang *et al*. [17]. Specifically, PRIM is a rule-set-guided scheduling algorithm that strives to improve energy efficiency on AMPs. At a high level, this scheme works as follows. Initially, when a thread is created, it is mapped to a random core type in the system in order to preserve load balance. Every so often, the scheduler randomly selects a certain number of thread pairs consisting of a thread running on a big core (T_B) and another thread running on a small core (T_S). For each randomly selected pair, the scheduler estimates whether swapping T_B with T_S would result in energy savings by means of a set of platform-specific rules.[3] If that is the case, the scheduler swaps both threads.

In the original work [17], the PRIM algorithm was simulated. To evaluate this algorithm on real asymmetric multicore hardware, we created an implementation for it in the Linux kernel. The evaluation of the PRIM scheme on a real OS and using actual asymmetric hardware is an important contribution of our work. In evaluating PRIM, we detected two important limitations of this scheduling scheme. First, it does not always minimize EDP, and is subject to throughput degradation, as we demonstrate in Section 5. Second, the inherent nature of the rules on which PRIM relies leads this scheduler to suboptimal thread-to-core mappings. Note that PRIM platform-specific rules do not quantify the actual energy savings resulting from a thread swap, but instead indicate whether a specific thread swap would be beneficial or not in terms of energy consumption. Therefore, the PRIM scheduler cannot tell whether there is a better candidate thread running

[3]Evaluating these platform-specific rules at run time requires the OS to gather threads' high-level performance metrics (such as the IPC or the LLC miss rate) by means of hardware counters.

on a small core T_{S2} such that swapping T_B with T_{S2} would result in higher energy savings than those resulting from swapping T_B with T_S. This issue, coupled with the fact that thread pairs are selected randomly by PRIM, often causes this approach to make suboptimal thread-to-core mappings. The EEF-Driven scheduler is not subject to these limitations.

It should be noted that our work focuses on general-purpose systems. Nevertheless, for embedded workloads, further optimizations are possible, since threads may exhibit more predictable execution patterns. In this context, Petrucci *et al*. [18] proposed a global optimization scheme that targets embedded thread sets with periodic characteristics. Their user-level scheduling proposal is able to determine energy-efficient thread assignments by leveraging an Integer Linear Programming model. This scheduler enforces thread-to-core mappings by imposing thread affinities via system calls. By contrast, we propose two light-weight OS-level general-purpose scheduling schemes that do not rely on any assumption about the workload characteristics.

Finally, we should also highlight that none of these works on energy consumption on AMPs analyzed the interrelationship between energy efficiency, system throughput and fairness, as we do here. Moreover, to carry out our experimental evaluation, we do not rely on simulation [17] or emulated asymmetry via frequency scaling [18], but instead we experiment with schedulers in an actual OS running on real asymmetric hardware.

4. DESIGN

The analytical study presented in Section 2 raises some important questions:

- How can the EEF-Driven scheduler be implemented on a real system, and how can it react to program phase changes at run time?
- How can we determine an application's EEF online on commercial off-the-shelf asymmetric multicores?
- Is it possible to create special knobs in the scheduler to make it possible for the user to trade fairness for throughput or energy efficiency?

We provide an answer for these questions in the following three sections, respectively.

4.1. The EEF-Driven scheduler

EEF-Driven assigns threads to big and small cores so as to preserve load balance in the AMP, and periodically readjusts thread-to-core mappings in accordance with the EEF of the various threads.

When a new thread enters the system, EEF-Driven assigns a *default value* of the EEF to it, since the actual value is unknown at that point.[4] The initial mapping of newly created threads is performed such that the load balance across the cores is preserved. As soon as the thread begins to run, the scheduler continuously monitors the IPC and other relevant high-level metrics using hardware performance counters, in order to estimate the thread's EEF at run time. The EEF estimation mechanism is described in Section 4.2.

As threads run, two things happen: EEFs for newly arrived threads become known, and EEFs of *old* threads may change (as they go through different program phases). The scheduler must map threads to cores according to their EEF, and, to that end, it triggers *event-driven migrations*. Overall, event-driven migrations ensure that the system adheres to the following two rules: (i) all threads on big cores have a higher EEF than the thread with maximum EEF running on a small core and (ii) load balance must be preserved. In order to enforce Rule 1, the scheduler must check that the thread with minimum EEF on big cores (T_{BC}) has a higher EEF than the thread with highest EEF on slow cores (T_{SC}). This rule may be broken either when a change in the EEF of a thread takes place or in the event that a thread transitions between a runnable and a non-runnable state (e.g. it blocks due to a page fault or due to synchronization). In the first scenario, the scheduler enforces the rule by swapping T_{BC} and T_{SC} when needed. In the second case, migrating one of the aforementioned threads to a different core type is enough to guarantee that the two rules of EEF-Driven hold true.

To simplify the enforcement of the two rules, the scheduler maintains per-core-type lists of runnable threads sorted by EEF to aid in the selection of the most appropriate thread(s) to be migrated (or swapped): T_{BC} and T_{SC}. For efficiency reasons, the big core list is kept sorted in an ascending order by EEF, while a descending order by EEF is preferred for the small-core thread list. As a result, finding the most appropriate swap candidate has constant complexity.

We found that reacting immediately to sudden changes in the EEF of a thread (e.g. during abrupt phase changes or spikes in the EEF) may cause premature and potentially costly migrations, which may degrade performance. To address this issue, the scheduler considers raw EEF estimations directly only in the event that a thread has entered a phase exhibiting stable behavior. For this task, we use a phase-detection mechanism inspired by that used in some variants of the HSP approach [6]. At a high level, this mechanism works as follows. The scheduler keeps a running average of the EEF values estimated over time for a thread. When the running average is updated, it is compared with the previous value of this average. If the difference between both values exceeds a given threshold, a phase transition is indicated.

[4] For this default value, we opted to choose the lowest EEF observed for SPEC CPU benchmarks on the platform. In doing so, threads with a relatively low estimated EEF and legitimately assigned to big cores are not relegated to small cores when new threads enter the system.

Two or more sampling intervals containing no indicated phase transition signal a stable phase. During stable phases the scheduler uses the last EEF estimate for the thread to make scheduling decisions. By contrast, when a thread goes through unstable EEF stages (frequent spikes), the scheduler uses the running average of the EEF values observed in the last few intervals. This makes it possible for EEF-Driven to mitigate the effects of oscillations and, in turn, to effectively reduce the number of thread migrations.

4.2. Determining the EEF

Some applications in a workload may exhibit a uniform EEF during the vast majority of its execution, whereas others go through different EEF phases over time. Figure 3 illustrates this fact by showing the EEF over time for two applications with different behavior.

To cater to the time-changing behavior of the applications, the EEF-Driven scheduler readjusts thread-to-core mappings dynamically based on the *current* EEF of the various threads. To make this possible, the OS scheduler must be equipped with a mechanism to obtain a thread's EEF over time. Designing such a mechanism is a challenging task, in part due to the fact that direct measurement of the EEF at run time is not possible. Recall that a thread's EEF is defined as the ratio of the thread's SF to its EPI rate on the big core (EPI_{big}). As stated in Section 3.1, measuring the SF directly is feasible, though not very effective, since it often leads to inaccurate values [7]. Unfortunately, measuring a thread's EPI_{big} directly when running together with other threads on the platform is typically not possible on a real system. Most commercial multicore systems that are equipped with energy registers or sensors, do not provide per-thread measurements, but instead system-wide measurements. Specifically, these registers provide energy (or power consumption) readings for different core clusters sharing a last-level cache, and for DRAM [20, 42].[5] With this support, the OS cannot isolate the individual contribution of each application to the aggregate DRAM or core cluster energy consumption in the context of a multi-program workload.

To overcome this limitation, the EEF-Driven scheduler determines a thread's EEF online by feeding a platform-specific prediction model with the value of various performance metrics gathered via hardware counters. Note that because the value of a performance metric (e.g. the IPC) may differ across core types, the scheduler actually relies on two models: one for predicting the EEF from big-core metrics and another for predicting it from small-core metrics. These two estimation models (used for all threads) are generated offline.

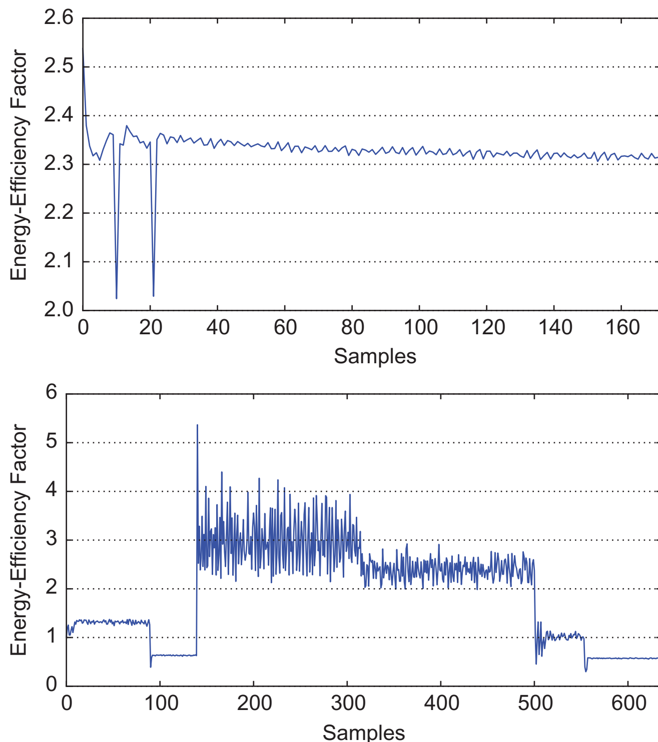

FIGURE 3. EEF over time observed for the `equake` (top) and `astar` (bottom) programs when running alone on the ARM Juno board.

4.2.1. *Building EEF estimation models*

To aid in the construction of the EEF estimation models, we used a variant of the *Phase-SF* methodology proposed in [19], which was originally devised to build SF estimation models. Overall, the methodology entails performing the following steps. We first pick a representative set of single-threaded applications (e.g. a subset of SPEC CPU benchmarks) and a comprehensive set of performance metrics that allow characterization of the microarchitectural and memory behavior of the various programs. We then run these benchmarks on big and small cores to monitor the performance metrics via performance monitoring counters (PMCs) using fixed instruction windows. Because gathering monitoring information from the entire execution of the benchmarks can be a time-consuming process, we collected information for a small number of instruction windows only. During the execution on the big core, we also measure the EPI_{big}. The gathering of this information entails sampling PMCs and energy registers in a fully coordinated fashion. By using the collected data, we identify coarse-grained program EEF phases by matching the various samples collected on both core types for contiguous instruction windows of the same application. (The EEF of a certain application's instruction window can be obtained from the IPC values collected on

[5]Most recent Intel (symmetric) multicore processors provide per-core energy readings, but still the DRAM energy consumption register reports the aggregate energy consumption.

both core types plus the EPI_{big} value observed for that instruction window.) Finally, we use the per-application EEF-phase data obtained in the previous step as input to the additive-regression engine provided by the WEKA machine-learning tool [43]. By using this engine, we generate two estimation models, enabling the prediction of the EEF from the big and from the small core. It should be noted that the additive-regression engineï»¿ automatically identifies irrelevant performance metrics (very low regression coefficients). In order to reduce the number of metrics that the scheduler needs to monitor at run time to obtain predictions, we discarded these irrelevant metrics in building the final estimation models.

Figure 4 illustrates the accuracy of the predictions obtained with the final EEF estimation models for both core types on the ARM Juno development board used for our experiments. The correlation coefficients for the estimation on the big and the small core are 0.97 and 0.95, respectively. In generating the models, we used performance and energy data from over 500 EEF phases from various benchmarks from the SPEC CPU suite. (In the experimental evaluation shown in the next section we used additional applications different from those used to generate the estimation models.) Specifically, Table 5 enumerates the set of performance metrics and associated hardware events that the estimation models depend upon.

4.2.2. *Obtaining EEF estimates at run time*

In order to determine a thread's EEF at run time, by using the estimation models presented in the previous section, the OS must continuously monitor the necessary performance metrics (Table 5) as a thread runs by means of hardware performance counters.

We should highlight that accessing performance counters directly from the core scheduler code would lead the implementation to be tied to specific processor models. To prevent this from happening, we have implemented both estimation models by means of a PMCTrack monitoring module [25]. The monitoring module (deployed in a loadable kernel module) is responsible for sampling hardware counters, and feeds the scheduler with per-thread EEF estimates at run time. In using this approach, the core scheduler implementation (inside the OS kernel) remains fully architecture independent. Extending the EEF-Driven scheduler to work on any asymmetric platform comes down to implementing the associated monitoring module.

4.3. ACFS-E: trading fairness for energy efficiency or throughput

Most scheduling algorithms that attempt to optimize throughput or improve energy efficiency—such as HSP [5, 6] or the EEF-Driven scheme—are subject to an important limitation: they deliver a fixed trade-off between fairness, throughput and energy consumption. At the same time, as our study in Section 2 reveals, these scheduling schemes are inherently unfair. This stems from the fact that optimizing throughput or energy consumption typically entails mapping a subset of threads in the workload to big cores and relegating the remaining threads to small cores for long periods of time. This makes it difficult to augment these scheduling schemes with quality-of-service support (e.g. by enabling the user to impose application priorities), or to implement scheduler *knobs* that allow the system administrator to trade one metric for another (e.g. fairness for throughput).

The ACFS scheduler [14] overcomes some of these issues. This scheduler seeks to optimize fairness on AMPs by evening out observed slowdowns across applications in the

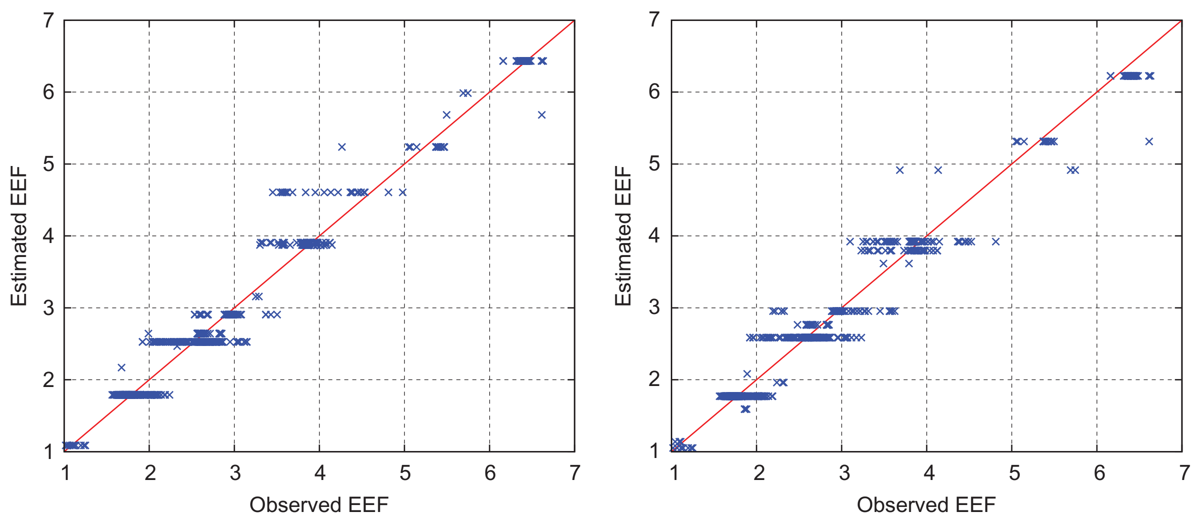

FIGURE 4. EEF prediction on the big (left) and the small (right) core for various program phases from SPEC CPU benchmarks running on the ARM Juno development board. Perfectly accurate estimations have all points on the diagonal line.

TABLE 5. Performance metrics and associated hardware events used to predict the SF and the EEF on the various cores of the ARM Juno development board.

Hardware events	Performance metrics	EEF (big)	EEF (small)	SF (big)	SF (small)
Instructions retired, processor cycles	Instructions per cycle	✓	✓	✓	✓
L2 (LLC) cache misses	LLC cache misses per 1 K instr.	✓	✓	✓	✓
L2 Data cache accesses	L2 Data cache accesses per 1 K instr.	✓	✓		
L1 data cache misses	L1 data cache misses per 1 K instr.		✓	✓	✓
Data memory access	Data memory accesses per 1 K instr.			✓	
Predictable branches speculatively executed	Predictable branch speculatively executed per 1 K instr.	✓		✓	
Conditional branch executed	Conditional branch executed per 1 K instr.		✓		✓
Mispredicted or not predicted branches speculatively executed	Mispredicted or not predicted branch speculatively executed per 1 K instr.			✓	✓
L1 instruction TLB misses	L1 ITLB misses per 1 M instr.	✓		✓	
`STALLS_A`: Counts every cycle there is an interlock that is not because of an Advanced SIMD or Floating-point instruction, and not because of a load/store instruction waiting for data to calculate the address in the AGU	`STALLS_A` per 1 K cycles.		✓		✓
`STALLS_B`: Counts every cycle the DPU IQ is empty and there is an instruction cache miss being processed	`STALLS_B` per 1 K cycles.		✓		✓
`STALLS_C`: Counts every cycle the DPU IQ is empty and there is an instruction micro-TLB miss being processed	`STALLS_C` per 1 K cycles.				✓

workload. To this end, it factors in threads' SFs when making scheduling decisions. ACFS also supports user-defined priorities, and is equipped with a knob, referred to as the *unfairness factor*, which enables the system administrator to trade fairness for throughput.

In this work, we extend ACFS with a mechanism that allows the user to trade fairness for energy efficiency. Prior to describing this extension, we first outline the basics of the original ACFS scheduler. To keep track of the relative progress of the various applications on the AMP, ACFS maintains a progress counter for each thread referred to as `amp_vruntime`. This counter tracks how much progress the thread has made thus far with respect to the progress that would have resulted from running it on a big core the whole time. 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:

$$\Delta_{\texttt{amp_vruntime}} = \frac{100 \cdot W_{\text{def}}}{S_{\text{core}} \cdot W_t}, \tag{10}$$

where W_t is the thread's weight, derived directly from the application priority (set by the user); W_{def} is the weight of applications with the default priority; and 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 application's current big-to-small speedup, which is estimated online. To enforce fairness, ACFS aims to even out the `amp_vruntime` across threads. To make this possible, it may need to perform thread swaps (migrations) between different core types every so often.

In synthetic scenarios like the one considered in Section 2 (multi-application workloads consisting constant-SF applications) and assuming perfect SF estimations, the base implementation of ACFS makes the same big-core cycle distribution across applications as the Opt-Unf theoretical scheduler [19]. Hence, ACFS provides the maximum throughput attainable for the optimal unfairness. For values of the *unfairness factor* greater than the default setting, higher throughput can be obtained at the expense of degrading fairness [19].

To enable the system administrator to trade fairness for energy efficiency (EDP), we augmented ACFS with the `EDP_factor` knob. Henceforth, we will refer to this variant of ACFS as *ACFS-E*. The proposed mechanism works as follows. 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 `EDP_factor` values greater than the default setting, the scheduler reduces the EDP at the expense of degrading fairness. Intuitively, making this possible comes down to gradually increasing the big-core share for those applications in the workload with a higher EEF while reducing the big-core share of the remaining ones. To this end, we factor in the `EDP_factor` when updating a thread's

`amp_vruntime` every tick. This entails replacing the thread's *static* priority or weight (W_t) in equation (10) with its *dynamic* weight (DW_t), which is defined as follows:

$$DW_t = W_t \cdot \left(1 + \frac{(\texttt{EDP_factor} - 1) \cdot (EEF_t - EEF_{min})}{EEF_{max} - EEF_{min}}\right),$$

where EEF_t denotes the thread's EEF, and EEF_{max} and EEF_{min} are the maximum and minimum EEFs observed among threads in the workload, respectively. Essentially, with this new definition of $\Delta_{\texttt{amp_vruntime}}$, the `amp_vruntime` of high-EEF threads increases at a slower pace than that of low-EEF threads, which results in a higher big-core share for high-EEF applications and, in turn, in higher EDP reductions. We have observed this trend by using the analytical model presented in Section 2. In Section 5.2, we experimentally analyze the effect of varying the `EDP_factor` using our implementation of ACFS-E in the Linux kernel. Our experiments reveal, among other things, that using a high value of the `EDP_factor` (5 in our experimental platform) enables ACFS-E to achieve comparable energy-efficiency figures to those obtained with the EEF-Driven scheduler, which strives to optimize the EDP.

In order for ACFS-E to update a thread's `amp_vruntime` every tick, the scheduler needs to determine the thread's current SF and EEF online. To obtain up-to-date EEF values on our experimental platform, we used the EEF estimation models presented in Section 4.2. To approximate a thread's SF on the big and on the small core, we built two estimation models by using the *Phase-SF* methodology presented in [19]. Figure 5 illustrates the accuracy of the predictions obtained with the SF estimation models for the ARM Juno development board. The correlation coefficients for the estimation observed on both core types are 0.95. Table 5 shows the set of performance metrics and associated hardware events that the OS must monitor at run time for each thread to obtain SF and EEF predictions on the various cores on our experimental platform.

5. EXPERIMENTAL EVALUATION

In our experiments, we compare the effectiveness of the EEF-Driven scheme with that of several previously proposed asymmetry-aware schedulers: HSP [5, 6], PRIM [17], ACFS [14] and RR [8]. We also experimented with the ACFS-E scheduler, presented in Section 4.3. We implemented all these schemes as a separate scheduling class in the Linux kernel v3.10. As discussed in Section 3.2, we omit the results of the Linux default scheduler (with and without the HMP patch [37]), as it does not ensure repeatable application completion times across multiple runs of the same workload.

All the schedulers considered (except for RR) rely on PMCs to function. Specifically, our implementation of HSP and ACFS determines threads' SFs online using the PMC-based estimation models presented in Section 4.3. EEF-Driven and ACFS-E also rely on PMCs to estimate thread's EEFs at run time by using the prediction models described in Section 4.2. Finally, the PRIM scheduler leverages a set of platform-specific PMC-based rules to determine whether a given thread swap brings energy savings or not. Note that this algorithm had been evaluated before via simulation only [17]. To be able to compare this scheme against our proposal, we created a real-world implementation of PRIM in the Linux kernel, and adapted the rules presented in [17] to the real asymmetric hardware platform we used. We should also highlight that to ensure that the core implementation of the various schedulers (included in the Linux kernel) remains platform independent,

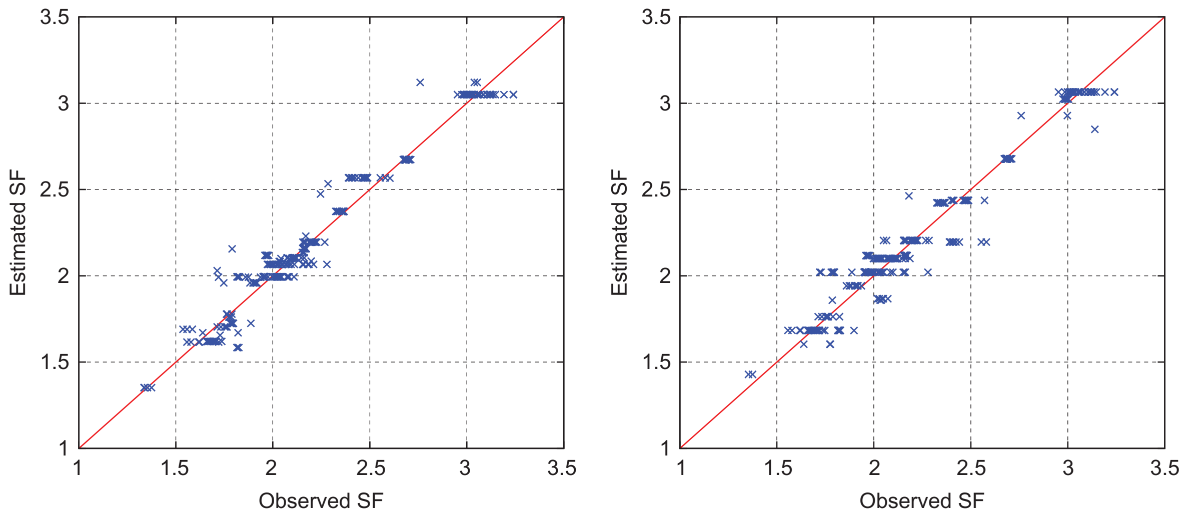

FIGURE 5. SF prediction on the big (left) and the small (right) core for various program phases from SPEC CPU benchmarks running on the ARM Juno development board.

the different PMC-based estimation models on which they depend were implemented inside a loadable kernel module using various PMCTrack monitoring modules [25]. Each monitoring module samples performance counters every 200 ms on a per-thread basis; this sampling interval was used in previous work on asymmetry-aware scheduling [18, 6]. We observed that the overhead associated with PMC sampling and determining thread's EEFs and SFs becomes negligible at this rate.

To evaluate the effectiveness of the various algorithms, we used the ARM Juno Development board [20], which integrates a 64-bit ARM big.LITTLE processor consisting of two *big* cores (Cortex A57) and four *small* (Cortex A53). Table 6 shows more information about this system. Each core of the big.LITTLE processor features a private L1 cache and shares a last-level (L2) cache with the other cores of the same type. On this asymmetric system, we experimented with two asymmetric configurations, referred to as *2B–4S* (two big cores and four small cores) and *1B–3S* (one big core and three small cores). The 1B–3S configuration enables us to study the behavior of the various scheduling schemes in a scenario where applications have to compete for a single big core with a 'private' 2 MB last-level cache.

Our evaluation targets multi-application workloads consisting of compute-intensive benchmarks from the SPEC CPU2006 and CPU2000 suites. We opted to use this kind of workloads to ensure a fair comparison against PRIM, HSP, RR and ACFS, as these schedulers were evaluated before using similar workloads [5, 8, 14, 17]. In all 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 that also employs compute-intensive workloads [5, 6, 18]. In each multi-application workload, 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 ASP and unfairness for the scheduler in question, by using the geometric mean of the completion times for each program. To measure the EDP for each workload (by using Equation (3)), we use the PMCTrack tool [25], which is equipped with support for gathering monitoring information from energy registers and performance counters available on the Juno board.

The remainder of this section is divided into two parts. In Section 5.1, we analyze the effectiveness of the EEF-Driven scheduler. In that section, we also discuss the results of the base implementation of ACFS [14], which (as stated in Section 4.3) behaves just like the ACFS-E scheduler with the default setting of the `EDP_factor` (1.0). In Section 5.2, we assess the impact of varying the `EDP_factor` knob under ACFS-E.

TABLE 6. Features of the ARM Juno development board.

Processor model (s)	Cortex A57	Cortex A53
Core count	2	4
Processor frequency	1.10 GHz	850 MHz
Pipeline	Out-of-order	In-order
Last-level cache (L2)	2 MB/16-way	1 MB/16-way
Main memory	8 GB DDR3 @ 800 MHz	

5.1. Evaluation of EEF-Driven

5.1.1. Workload selection

Table 7 shows the composition of the 20 workloads that we analyzed in detail under the various schedulers considered. The first 10 program mixes, consisting of six applications each were run on the 2B–4S configuration; the last 10 mixes (M11-M20), with four applications each, were run on 1B–3S. In building the workloads, we generated random program mixes (with four or six applications each) by combining 18 SPEC benchmarks that cover a wide spectrum of SF and EEF values. From these workloads, we filtered out those application mixes where the HSP and the EEF-Driven schemes perform the same big-core cycle distribution for the vast majority of the execution. For these workloads, the applications with the highest overall SFs also happen to be those with the highest EEF values. Thus, optimizing throughput (the HSP approach) also leads to optimization of the EDP. This type of program mixes does not showcase the potential of energy-aware schedulers, as the HSP and EEF-Driven yield very close ASP, unfairness and energy-efficiency values (differences in a 1% range) in these scenarios. By contrast, for workloads in Table 7 HSP and EEF-Driven perform very differently. The various programs in

TABLE 7. Multi-application workloads.

Name	Applications
M1	`equake,soplex,vortex,perlbmk,povray,gobmk`
M2	`equake,gamess,hmmer,perlbench,gzip,gobmk`
M3	`galgel,equake,hmmer,povray,mgrid,gobmk`
M4	`galgel,gamess,hmmer,povray,perlbench,gobmk`
M5	`equake,gamess,hmmer,gobmk,crafty,sixtrack`
M6	`galgel,equake,gamess,hmmer,sixtrack,povray`
M7	`galgel,equake,hmmer,bzip2,perlbench,h264ref`
M8	`gamess,art,gobmk,crafty,sixtrack,vortex`
M9	`gamess,art,bzip2,gobmk,sixtrack,vortex`
M10	`soplex,bzip2,perlbench,gzip,h264ref,gobmk`
M11	`equake,hmmer,sixtrack,povray`
M12	`equake,hmmer,povray,astar`
M13	`galgel,hmmer,povray,mgrid`
M14	`equake,gamess,hmmer,perlbench`
M15	`equake,vortex,povray,h264ref`
M16	`galgel,hmmer,crafty,perlbench`
M17	`soplex,vortex,povray,h264ref`
M18	`art,vortex,perlbmk,povray`
M19	`art,perlbmk,povray,h264ref`
M20	`art,perlbmk,sixtrack,povray`

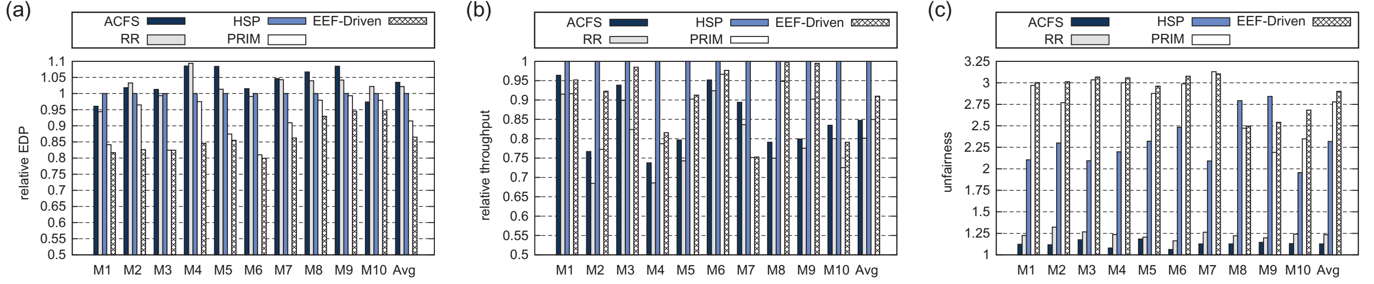

FIGURE 6. Results for workloads M1–M10 running on 2B-4S. (**a**) Relative EDP, (**b**) Relative throughput, (**c**) Unfairness.

each workload shown in the table are listed in descending order by its overall SF (observed throughout the execution). Hence, the programs displayed first enjoy a higher big-core share under the HSP scheduler than the rest of the programs.

5.1.2. Discussion

Figure 6a–c display the EDP, throughput (ASP) and unfairness for workloads M1–M10 running on 2B–4S under the various scheduling algorithms. The reported EDP and ASP values have been normalized with respect to the results of the HSP scheduler.

The experimental results exhibit similar trends to those extracted from our theoretical study (Section 2). Clearly, optimizing one metric may lead to substantial degradation of another metric. As is evident, the HSP scheduler, which strives to optimize throughput achieves the best ASP values across the board. Conversely, the EEF-Driven scheduler obtains the best EDP results (the lower, the better) for all workloads. HSP and EEF-Driven, however, may degrade the EDP and the throughput, respectively. In particular, HSP can experience up to a 20% degradation in EDP with respect to EEF-Driven, in exchange for a 2.5% increase in throughput (e.g. for the M6 workload). Conversely, EEF-Driven may also obtain modest EDP improvements with respect to HSP in some cases (5% for M10) at the expense of significant performance degradation (21% for M10). Notably, the relative throughput degradation for the M10 workload is even higher under the PRIM scheme (27.5%), which achieves only a 2.5% reduction in EDP compared to HSP for that workload. Regarding fairness, the results reveal that RR and ACFS, which are fairness aware, are the only schedulers that deliver decent unfairness figures (close to one). Delivering fairness, however, comes at the cost of EDP and throughput degradation compared to other schemes that seek to optimize throughput (HSP) or aim to improve energy efficiency (EEF-Driven and PRIM).

We now proceed to analyze the results of the energy-aware schedulers considered: PRIM and the EEF-Driven scheduler (our proposal). As discussed in Section 3.3, PRIM strives to improve energy efficiency by performing thread swaps that lead to energy savings. In doing so, threads with a lower EPI ratio on the big core (EPI_{big}) are typically mapped to big cores for longer periods of time than other threads. We found that for some workloads (such as M1, M5 or M6), this scheduler performs very close big-core cycle distributions to those enforced by EEF-Driven, hence the similar results observed for these workloads. In these scenarios, programs with the lowest EPI_{big} value also exhibit the highest EEF values. By contrast, in the rest of workloads, EEF-Driven clearly improves upon PRIM in both EDP (up to a 15% reduction for M2) and throughput (up to a 20% increase -M2-). This difference primarily stems from the fact that in these scenarios, several programs exhibit a similar EPI_{big} but different EEF values (due to a different SF). In the event that two threads go through program phases with similar EPI_{big} values, the thread with the highest SF is mapped preferentially to the big core by EEF-Driven. Failing to consider the ratio of the SF to the EPI_{big} (aka. the EEF) leads PRIM to performing thread-to-core mappings that do not optimize the EDP and, at the same time, degrade throughput significantly. We also observed that the random nature of the thread swaps performed by PRIM also leads to suboptimal thread-to-core mappings for short periods of time (as explained in Section 3.3), which results in worse EDP values compared to EEF-Driven.

As for the fairness-aware schedulers considered, the results reveal that ACFS delivers better throughput and fairness than RR across the board. This stems from the fact that ACFS factors in the SF when making scheduling decisions whereas RR does not. The results also illustrate that both schedulers achieve significantly worse EDP values—up to a 25% higher (M4 workload)—than those obtained by EEF-Driven.

We now turn our attention to the results for the M11–M20 workloads running on the 1B–3S configuration. The relative EDP, relative ASP and unfairness for the various workloads and schedulers are reported in Fig. 7a–c, respectively. The results showcase similar trends to those observed for the 2B–4S configuration. Nevertheless, two important observations are in order. First, EEF-Driven brings a higher average reduction in EDP with respect to HSP on this configuration (20%) compared to that that observed on 2B–4S (13.5%). At the same time, the average throughput degradation relative to

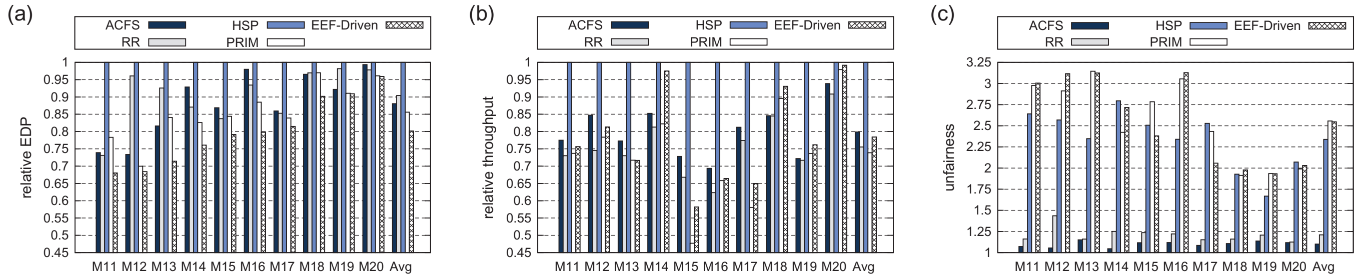

FIGURE 7. Results for workloads M11–M20 running on 1B-3S. (**a**) Relative EDP, (**b**) Relative throughput, (**c**) Unfairness.

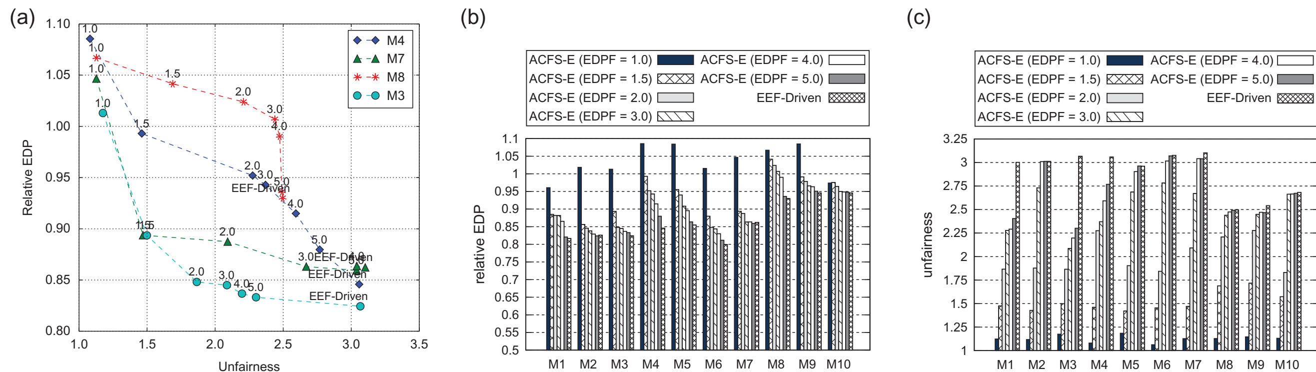

FIGURE 8. Unfairness vs. relative EDP for different values of the `EDP_factor` (EDPF) under the ACFS-E scheduler. (**a**) Unfairness vs. Relative EDP for selected workloads, (**b**) Relative EDP, (**c**) Unfairness.

HSP also increases from 9% to 21.5%. Essentially, for workloads M11–M20, the application listed first in the corresponding row of Table 7 (e.g. `equake` for M11) is the one that enjoys a higher big core share under HSP. This stands in contrast with what the EEF-Driven scheme does: it maps the application listed in the second place (e.g. `hmmer` for M11) to the big core for the vast majority of the execution due to its higher EEF. If a second big core were available (as on 2B–4S), both schedulers would grant a substantial big-core share to both applications, which would lead to closer EDP and ASP values. Second, fairness-aware schedulers achieve lower EDP values than HSP across the board; this was not the case on the 2B–4S configuration. Intuitively, the big-core share of all applications (including those with a high EPI_{big} value) increases under RR and ACFS on 2B–4S due to the additional big core; this results in significantly higher energy consumption and EDP on 2B–4S than on 1B–3S.

5.2. Effectiveness of the ACFS-E scheduler

All the scheduling schemes evaluated in the previous section deliver a fixed trade-off between fairness, throughput and energy efficiency. In the quest of a more flexible and configurable scheduling scheme, we designed the ACFS-E scheduler (described in Section 4.3). This fairness-aware scheduler is equipped with two knobs, referred to as `EDP_factor` and `unfairness_factor`. When these parameters are set to the default setting (1.0), ACFS-E behaves as the original ACFS scheduler [14, 19], which strives to deliver the maximum ASP (throughput) attainable for the optimal unfairness. When a non-default value of either of these parameters is established,[6] a specific dynamic priority scheme is engaged, which adjusts a thread's priority based on either its SF (for the `unfairness_factor`) or its EEF (for the `EDP_factor`). As discussed in Section 4.3, the two knobs empower the system administrator with a means to provide a configurable balance between fairness and EDP or system throughput on the AMP.

Figure 8a shows how the choice of the `EDP_factor` affects the unfairness and the EDP for four selected workloads from Table 7 (running on the 2B–4S configuration). The results reveal that the default and lowest possible setting for the `EDP_factor` provides the best fairness figures, whereas higher values of this parameter lead to reductions in the EDP at the expense of fairness degradation. Notably, the results also illustrate that, as we gradually increase the

[6] It is not possible to set values different than the default setting for both parameters simultaneously under ACFS-E.

`EDP_factor`, the EDP and unfairness numbers under ACFS-E become closer to those of the EEF-Driven scheduler, which optimizes the EDP.

For the sake of completeness, Fig. 8b and c illustrates how the EDP and unfairness (respectively) vary for the full set of six-application workloads shown in Table 7 (M1–M10) when using values of the `EDP_factor` ranging between 1 and 5. As is evident, the same behavior observed for the selected workloads is also observed for the remaining program mixes: the higher the value of the `EDP_factor`, the lower the EDP and the higher the unfairness. The results also reveal that setting the `EDP_factor` to 5 enables ACFS-E to deliver a relative EDP value within a 1% range of the EEF-Driven scheduler for all the workloads explored, except for M4. Notably, we also tried increasing the `EDP_factor` beyond 5 for the M4 workload, and found that ACFS-E eventually (`EDP_factor`$\approx$6.5) achieves a very similar big-core cycle distribution among applications as that performed by EEF-Driven, hence leading both schedulers to very close EDP and unfairness figures (in a 1% range).

The results in Fig. 8b and c also illustrate that when the `EDP_factor` is set to 5, ACFS-E is capable to obtain better unfairness numbers than EEF-Driven for workloads M1, M3, M4 and M9, and, at the same time, it reaps comparable energy-efficiency results. This behavior stems from the fact that these workloads include at least three programs exhibiting a relatively high EEF and SF values for the vast majority of the execution, such as `vortex`, `galgel` or `hmmer`. On the AMP configuration used, the EEF-Driven scheduler devotes the two available big cores to running the two applications in the workload with the greatest EEF values, while relegating the rest to small cores. Conversely, in this scenario, ACFS-E grants a non-negligible big-core share to all high-SF high-EEF programs, which contributes to reducing unfairness and also delivers good energy-efficiency figures. Despite the striking reduction in EDP achieved by ACFS-E when increasing the `EDP_factor`, this scheme clearly yields high unfairness values when `EDP_factor` $\geq$ 2.

Unlike the `EDP_factor`, the `unfairness_factor` was already included in the base implementation of the ACFS scheduler. In [14], we demonstrated experimentally that this parameter provides a configurable balance between fairness and the system throughput on the Intel QuickIA prototype [3]. The results in Fig. 9a demonstrate that this parameter proves effective on the ARM Juno board as well. Specifically, the chart illustrates the impact of varying the `unfairness_factor` on fairness and throughput for the four selected workloads considered earlier. We observe that, in general, higher values of the `unfairness_factor` lead to throughput gains at the expense of degrading fairness. In addition, the results reveal that the higher the value of this parameter, the closer the ASP value provided by ACFS-E is to that of the HSP scheduler, which strives to optimize throughput.

Figure 9b and c illustrates how throughput and unfairness (respectively) vary for the M1–M10 workloads shown in Table 7 when using values of the `unfairness_factor` (UF) ranging between 1 and 2 in steps of 0.2. Similar trends observed for the selected program mixes are observed for the remaining workloads as well: the higher the setting of the UF, the closer the behavior of the ACFS-E scheduler is to that of HSP. For the vast majority of workloads, setting the UF value to 2 makes it possible for ACFS-E to achieve throughput figures in a 4% range to those of HSP. For a few workloads, such as M4 or M8, increasing the unfairness factor beyond 2 is necessary for ACFS-E to get that close to HSP in terms of throughput. Specifically, we tried increasing the UF further and found that in order for ACFS-E to perform in a 4% range of HSP under the M4 and M8 workloads, UF values as high as 3.6 and 3, respectively, must be used.

The results in Fig. 9b and c also reveal that for most workloads, higher values of the `unfairness_factor` lead to throughput gains coupled with fairness degradation. Contrary to our initial expectations, this is not exactly the observed behavior for workloads M1, M3 and M6; for these program mixes, ACFS-E is capable to deliver throughput gains relative

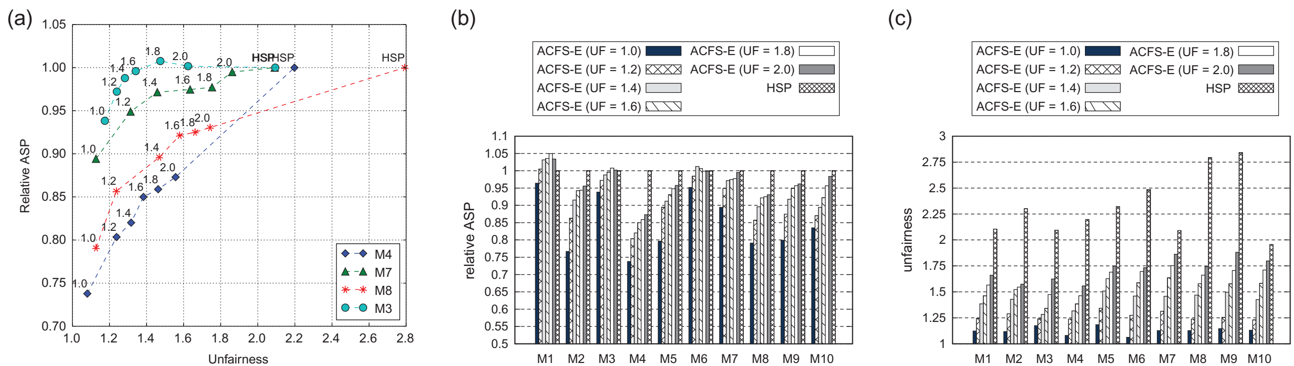

FIGURE 9. Unfairness vs. relative throughput (ASP) for different values of the `unfairness_factor` (UF) under ACFS-E. (**a**) Unfairness vs. Relative ASP for selected workloads, (**b**) Relative throughput, (**c**) Unfairness.

to the HSP scheduler (up to a 5% increase under M1). Moreover, the maximum throughput value observed is not reached when using the highest explored value of the `unfairness_factor`; instead, throughput drops when increasing the `unfairness_factor` beyond a certain point (e.g. 1.8 for M1 and 1.4 for M6). We found that this behavior comes from shared-resource contention effects that become apparent in the M1, M3 and M6 program mixes, which include several memory-intensive and cache-sensitive applications, such as `equake`, `soplex` and `galgel`.

To fully understand the results for the M1, M3 and M6 workloads, it is worth highlighting that none of the scheduling algorithms analyzed in this work (some of them proposed by other authors) deals with shared-resource contention effects. The aforementioned memory-bound applications present in these workloads happen to be the ones that derive the highest big-to-small speedup in the program mix. The benefit that these applications experience from the big cores comes in part due to the fact that this core type features a larger shared L2 cache than the small core (see Table 6 for details). Unfortunately, when the OS scheduler maps two memory-bound programs on the big cores simultaneously, cores compete with each other for space in the shared L2 cache as well as for bus bandwidth, which gives rise to non-negligible performance degradation for both applications, and in turn degrades system throughput.[7] Specifically, we found that under workloads M1, M3 and M6, the HSP scheduler maps memory-bound applications to big cores simultaneously for longer periods of time than ACFS-E when using the default setting of the `unfairness_factor` (1.0). As we increase the `unfairness_factor`, the ACFS-E scheduler grants a higher big-core share to HSP applications (the memory-bound programs in this case). Clearly, a modest increase of the big-core share of HSP memory-intensive applications results at first in higher performance for these application and, in turn, in substantial throughput gains. However, increasing too much the amount of time that these applications spend on the big cores (what happens as a result of increasing the `unfairness_factor` beyond a certain point), lead HSP memory-intensive applications to be mapped simultaneously to big cores more often, which backfires by degrading the performance of individual applications. This observation suggests that for workloads that feature multiple HSP memory-intensive applications taking shared-resource contention effects into consideration when making scheduling decisions may help improving system throughput even further on AMPs. Making ACFS-E aware of shared-resource contention issues constitutes a very promising avenue for future research.

[7]We also observed that contention on the L2 cache shared among little in-order cores is not so high, leading to much lower (and more uniform) performance degradation across applications in the workloads we explored. We hypothesized that this smaller degree of cache contention has to do with the fact that an in-order core cannot handle multiple outstanding cache misses; this leads to a smaller rate of LLC requests per cycle.

To sum up, our study has demonstrated that the `EDP_factor` and `unfairness_factor` knobs enable us to gradually improve energy efficiency or system throughput under ACFS-E when fairness constraints are relaxed. Moreover, by using high values for these parameters as those used in our experiments, ACFS-E can be configured to approximate the behavior of the EEF-Driven or HSP schemes, which optimize energy efficiency and throughput, respectively. Hence, the main takeaway from our study is that ACFS-E constitutes a flexible and versatile scheme, as it enables the system administrator to pursue several optimization goals with a single scheduling algorithm.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have carried out an analytical and experimental analysis that showcase the interrelationship between throughput, fairness and energy efficiency on AMPs in the context of OS scheduling. Although several schedulers that strive to optimize some of these aspects separately were proposed before [5, 6, 14, 17], no previous work had illustrated the impact that optimizing one of the aforementioned aspects alone has in the other two.

To fill this gap, we created a theoretical model making it possible to find the optimal schedule for the various optimization goals in diverse synthetic scenarios. To this end, we augmented the theoretical model presented in [11] with the ability to estimate the EDP analytically. Our analytical study enabled us to draw three major insights. First, optimizing the EDP on asymmetric multicores always leads to unfairness, and may also come at the expense of substantial throughput degradation. Second, in most cases, delivering fairness entails sacrificing throughput and energy efficiency to a great extent. Third, the schedule that ensures the maximum throughput attainable for the optimal (lowest) EDP value, can be approximated by leveraging an application's EEF, which is defined as the ratio of an application big-to-small speedup (aka SF) to the EPI rate achieved on the big core. Specifically, substantial reductions in the EDP can be accomplished by dedicating big cores to running applications in the workload that exhibit the highest EEF values.

By leveraging the EEF metric, we designed two novel energy-aware OS-scheduling algorithms for AMPs: EEF-Driven and ACFS-E. We implemented these algorithms in the Linux kernel and evaluated their effectiveness on a real asymmetric system that features an ARM big.LITTLE multicore processor. An extensive comparison with existing schemes [5, 6, 14, 17] was performed. Our experimental results reveal that the EEF-Driven scheduler reduces the EDP and improves throughput substantially compared to a state-of-the-art energy-aware scheduler [17]. We also demonstrated that asymmetry-aware schedulers that strive to optimize one metric alone (e.g. throughput) always deliver a fixed trade-off between system throughput, fairness and energy efficiency on AMPs, and so

these schemes may not provide a comprehensive support in general-purpose OSes. Our experimental analysis demonstrates that ACFS-E constitutes a more versatile approach as it can be configured to optimize any of the aforementioned three metrics, and, at the same time, enables the user to trade fairness for energy efficiency or throughput in scenarios where fairness constraints are relaxed.

An interesting direction for future work is to incorporate shared-resource contention awareness into asymmetry-aware algorithms like the ones studied in this paper. We strongly believe that this would be a first step towards designing all-encompassing scheduling algorithms for asymmetric multicore systems.

ACKNOWLEDGEMENTS

This work has been supported by the EU (FEDER) and the Spanish MINECO, under Grants TIN 2015–65277-R and TIN2012–32180, as well as by the HIPEAC-4 European Network of Excellence. Juan Carlos Saez has been also supported by a Santander postdoctoral mobility grant.

REFERENCES

[1] Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P. and Farkas, K. I. (2004) Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *Proc. ISCA 04*, Munich, Germany, June 19–23, pp. 64–75. IEEE Computer Society, Washington, DC, USA.

[2] Samsung (2012). Benefits of the big.LITTLE architecture. http://www.samsung.com/semiconductor/minisite/Exynos/data/benefits.pdf (accessed April 19, 2017).

[3] Chitlur, N. *et al.* (2012) QuickIA: Exploring Heterogeneous Architectures on Real Prototypes. *Proc. HPCA 12*, New Orleans, LA, February 25–29, pp. 1–8. IEEE Computer Society, Washington, DC, USA.

[4] Mittal, S. (2016) A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv.*, **48**, 45:1–45:38.

[5] Koufaty, D., Reddy, D. and Hahn, S. (2010) Bias Scheduling in Heterogeneous Multi-core Architectures. *Proc. Eurosys 10*, Paris, France, April 13–16, pp. 125–138. ACM, New York.

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

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

[8] Becchi, M. and Crowley, P. (2006) Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. *Proc. CF 06*, Ischia, Italy, May 2–5, pp. 29–40. ACM, New York.

[9] Annavaram, M., Grochowski, E. and Shen, J. (2005) Mitigating Amdahl's Law through EPI Throttling. *Proc. ISCA 05*, Wisconsin, USA, June 4–8, pp. 298–309. IEEE Computer Society, Washington, DC, USA.

[10] Joao, J. A., Suleman, M. A., Mutlu, O. and Patt, Y. N. (2012) Bottleneck Identification and Scheduling in Multithreaded Applications. *Proc. Seventeenth Int. Conf. Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, pp. 223–234. ACM.

[11] Saez, J. C., Pousa, A., Castro, F., Chaver, D. and Prieto-Matias, M. (2014) Exploring the Throughput-Fairness Trade-off on Asymmetric Multicore Systems. *Proc. Euro-Par 14: Parallel Processing Workshops*, Porto, Portugal, August 25–26, pp. 326–337. Springer, Berlin.

[12] Mutlu, O. and Moscibroda, T. (2007) Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. *Proc. MICRO '07*, Chicago, IL, December 1–5, pp. 146–160. IEEE Computer Society Washington, DC, USA.

[13] Ebrahimi, E., Lee, C. J., Mutlu, O. and Patt, Y. N. (2010) Fairness Via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. *Proc. ASPLOS 10*, Pittsburgh, PA, March 13–17, pp. 335–346. ACM, New York.

[14] Saez, J. C., Pousa, A., Castro, F., Chaver, D. and Prieto-Matias, M. (2015) ACFS: A Completely Fair Scheduler for Asymmetric Single-ISA Multicore Systems. *Proc. 30th Annual ACM Symposium on Applied Computing*, New York, NY, USA SAC '15, pp. 2027–2032. ACM.

[15] Li, T., Brett, P., Knauerhase, R. and Koufaty, D. (2010) Operating System Support for Overlapping-ISA Heterogeneous Multi-Core Architectures. *Proc. HPCA 10*, Bangalore, India, January 9–14, pp. 1–12. IEEE Computer Society Press, Los Alamitos, CA.

[16] Van Craeynest, K., Akram, S., Heirman, W., Jaleel, A. and Eeckhout, L. (2013) Fairness-Aware Scheduling on Single-ISA Heterogeneous Multi-Cores. *Proc. PACT 13*, Edinburgh, Scotland, September 7–11, pp. 177–187. ACM, New York.

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

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

[19] Saez, J. C., Pousa, A., Castro, F., Chaver, D. and Prieto-Matias, M. (2017) Towards completely fair scheduling on asymmetric single-ISA multicore processors. *J. Parallel Distrib. Comput.*, **102**, 115–131.

[20] ARM (2014). ARM juno development board. http://www.arm.com/products/tools/development-boards/versatile-express/juno-arm-development-platform.php (accessed November 16, 2015).

[21] Gabor, R., Weiss, S. and Mendelson, A. (2006) Fairness and Throughput in Switch on Event Multithreading. *Proc. MICRO 06*, Orlando, FL, December 9–13, pp. 149–160. IEEE Computer Society Washington, DC, USA.

[22] Horowitz, M., Indermaur, T. and Gonzalez, R. (1994) Low-Power Digital Design. *Proc. IEEE Symposium on Low Power Electronics*, San Diego, CA, October 10–12, pp. 8–11.

[23] Gonzalez, R. and Horowitz, M. (1996) Energy dissipation in general purpose microprocessors. *IEEE J. Solid-State Circuits*, **31**, 1277–1284.

[24] Blagodurov, S., Zhuravlev, S. and Fedorova, A. (2010) Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, **28**, 8:1–8:45.

[25] Saez, J. C., Pousa, A., Rodriíguez-Rodriíguez, R., Castro, F. and Prieto-Matias, M. (2017) PMCTrack: delivering performance monitoring counter support to the OS scheduler. *Comput. J.*, **60**, 60.

[26] Annamalai, A., Rodrigues, R., Koren, I. and Kundu, S. (2013) An Opportunistic Prediction-Based Thread Scheduling to Maximize Throughput/Watt in AMPs. *Proc. 22nd Int. Conf. Parallel Architectures and Compilation Techniques*, September, pp. 63–72.

[27] Hardkernel (2016). Odroid XU4 board. http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu4 (accessed June 22, 2016).

[28] Kumar, R. *et al*. (2003) Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. *Proc. of MICRO 36*.

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

[30] Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P. and Emer, J. (2012) Scheduling Heterogeneous Multi-Cores Through Performance Impact Estimation (PIE). *Proc. ISCA 12*, Portland, OR, June 9–13, pp. 213–224. IEEE Computer Society Washington, DC, USA.

[31] Pricopi, M., Muthukaruppan, T. S., Venkataramani, V., Mitra, T. and Vishin, S. (2013) Power-Performance Modeling on Asymmetric Multi-Cores. *Proc. CASES 13*, Montreal, Canada, 29 September–4 October, pp. 15:1–15:10. IEEE Press Piscataway, NJ, USA.

[32] Joao, J. A., Suleman, M. A., Mutlu, O. and Patt, Y. N. (2013) Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs. *Proc. ISCA 13*, Tel-Aviv, Israel, June 23–27, pp. 154–165. ACM, New York.

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

[34] Jibaja, I., Cao, T., Blackburn, S. M. and McKinley, K. S. (2016) Portable Performance on Asymmetric Multicore Processors. *Proc. 2016 Int. Symposium on Code Generation and Optimization*, New York, NY, USA, pp. 24–35. ACM.

[35] Markovic, N., Nemirovsky, D., Milutinovic, V., Unsal, O., Valero, M. and Cristal, A. (2015) Hardware Round-Robin Scheduler for Single-ISA Asymmetric Multi-Core. *Euro-Par 2015*, pp. 122–134. Springer.

[36] Petrucci, V., Loques, O. and Mossé, D. (2012) Lucky Scheduling for Energy-Efficient Heterogeneous Multi-Core Systems. *Proc. USENIX HotPower 12*, Hollywood, CA, 7 October, pp. 7–7. USENIX Association Berkeley, CA, USA.

[37] Rasmussen, M. (2012). Task placement for heterogeneous MP systems. https://lwn.net/Articles/517250/ (accessed July 06, 2016).

[38] Kim, M., Noh, S., Huh, S. and Hong, S. (2015) Fair-Share Scheduling for Performance-Asymmetric Multicore Architecture Via Scaled Virtual Runtime. *2015 IEEE 21st Int. Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 60–69. IEEE.

[39] Kim, C. and Huh, J. (2016) Fairness-Oriented OS Scheduling Support for Multicore Systems. *Proc. 2016 Int. Conf. Supercomputing*, New York, NY, USA ICS '16, pp. 29:1–29:12. ACM.

[40] Mogul, J. C., Mudigonda, J., Binkert, N., Ranganathan, P. and Talwar, V. (2008) Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, **28**, 26–41.

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

[42] ARM (2014). CoreTile Express development board. http://www.arm.com/products/tools/development-boards/versatile-express/coretile-express.php (accessed March 02, 2015).

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