Exploring the Throughput-Fairness Trade-off on Asymmetric Multicore Systems

Juan Carlos Saez[1], Adrian Pousa[2], Fernando Castro[1], Daniel Chaver[1], and Manuel Prieto-Matías[1]

[1] Computer Science School, Complutense University, Madrid, Spain
{jcsaezal,fcastror,dani02,mpmatias}@ucm.es
[2] Instituto de Investigacion en Informatica LIDI, UNLP, Argentina
apousa@lidi.info.unlp.edu.ar

Abstract. Symmetric-ISA (instruction set architecture) asymmetric-performance multicore processors (AMPs) were shown to deliver higher performance per watt and area than symmetric CMPs (Chip Multi-Processors). Previous work has shown that this potential of AMP systems can be realizable thanks to the OS scheduler. Existing scheduling schemes that deliver fairness and priority enforcement on AMPs do not cater to the fact that applications in a multiprogram workload may derive different benefit from using fast cores in the system. As a result, they are likely to perform thread-to-core mappings that degrade the system throughput. To address this limitation, we propose Prop-SP, a scheduling algorithm that aims to improve the throughput-fairness trade-off on AMPs. Our evaluation on real hardware, and using scheduler implementations on a general-purpose OS, reveals that Prop-SP delivers a better throughput-fairness trade-off than state-of-the-art schedulers for a wide variety of multi-application workloads.

Keywords: asymmetric multicore, scheduling, operating systems.

1 Introduction

Single-ISA asymmetric CMPs combine several core types with the same instruction-set architecture but different features such as clock frequency or microarchitecture. Previous work has demonstrated that asymmetric designs lead to a more efficient die area usage and a lower power consumption than symmetric CMPs [12]. Notably, combining just two core types simplifies the design and is enough to obtain most benefits from AMPs [13]. Major hardware players appear to be following this trend, as suggested by the recent ARM big.LITTLE processor [2] or the Quick-IA Intel prototype system [5].

Despite their benefits, AMPs pose significant challenges to the system software. One of the main challenges is to efficiently distribute fast-core cycles among the various applications running on the system. This task can be accomplished by the OS scheduler [18,11] or by the VM hypervisor on virtual environments [14]. Most existing proposals have focused on maximizing the system throughput

[13,21,18,11]. To make this possible the scheduler needs to map to fast cores predominantly application threads that use those cores efficiently since they derive performance improvements (speedup) relative to running on slow cores [13]. Further throughput gains can be achieved by using fast cores to accelerate sequential phases of parallel programs [19,10].

Other important goals such as delivering fairness or priority enforcement on AMPs have drawn less attention from the research community. Previously-proposed OS-level schemes that deliver fairness on AMPs attempt to allocate a *fair* heterogeneous CPU share to the various applications. This can be accomplished by fair-sharing fast cores among applications [3,18] or by factoring in the computational power of the various cores when performing CPU accounting [15]. None of these techniques, however, exploit the fact that applications in a multiprogram workload may derive different benefit from using the fast cores in the AMP. For this reason, assigning the same heterogeneous CPU share to equal-priority applications does not ensure an *even slowdown across applications due to sharing the AMP* [20]. Moreover, not taking into account the diversity in applications' relative speedups when making scheduling decisions on AMPs may also lead to degrading the system throughput [3,18].

To address these shortcomings, we propose Prop-SP, a novel scheduling algorithm that delivers priority enforcement on AMPs and strives to even out the slowdown experienced by equal-priority applications. Our proposal delivers high system throughput without requiring hardware support nor changes in the applications. We qualitatively and quantitatively compare Prop-SP with state-of-the-art schedulers, such as A-DWRR [15] and CAMP [18]. Our experimental analysis reveals that Prop-SP improves the throughput-fairness trade-off for a broad spectrum of multi-application workloads.

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

2 Motivation

We now present an analytical study regarding the system throughput and fairness delivered by previously proposed scheduling algorithms for AMPs. Our analysis demonstrates that existing schedulers that seek to optimize one metric degrade the other significantly, thus achieving unacceptable tradeoffs.

To assess *system throughput* we avoided metrics depending on *instructions per cycle* (IPC) or *instructions per second* (IPS) since they can be misleading to evaluate the performance of multithreaded programs [1]. As such, we opted to use a metric depending on *completion time* instead. In particular, we found that the *Aggregate Speedup* captures differences in throughput caused by diverse asymmetry-aware schedulers considerably better than other metrics proposed for CMPs, such as STP [7]. The *Aggregate Speedup* is defined as follows:

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

Table 1. Synthetic workloads

Workload	$\mathbf{SF}_1$	$\mathbf{SF}_2$	$\mathbf{SF}_3$	$\mathbf{SF}_4$
W1	3.4	3.4	1.2	1.2
W2	3.4	3.4	2.3	2.3
W3	2.3	2.3	1.9	1.9
W4	3.4	3.4	2.7	2.7
W5	3.4	3.4	3.4	3.4
W6	2.5	2.1	1.6	1.2
W7	3.0	2.1	2.1	2.1
W8	3.4	3.0	2.5	2.1
W9	2.9	2.5	2.1	1.2

Table 2. Analytical formulas to approximate the aggregate speedup and unfairness for a workload consisting of n applications running simultaneously under a given thread scheduler.

Metric	Definition
Agreggate Speedup	$\sum_{i=1}^{n}\left(\frac{1}{\frac{f_i}{SF_i}+(1-f_i)}-1\right)$
$Slowdown_{app}$	$f_{app}+SF_{app}\cdot(1-f_{app})$
Unfairness	$\frac{MAX(Slowdown_1,...,Slowdown_n)}{MIN(Slowdown_1,...,Slowdown_n)}$

where n is the number of applications in the workload, $CT_{slow,i}$ is the completion time of application i when it runs alone in the system and uses slow cores only, and $CT_{sched,i}$ is the completion time of application i under a given scheduler.

Regarding *fairness*, previous works have employed diverse definitions. Some of them define a scheme to be fair if it assigns the same CPU share to equal-priority threads [15]. Others consider a scheme as fair if equal-priority applications suffer the same slowdown due to sharing the system with respect to the situation in which the whole system is available to each application [8,16,6]. The latter definition is more suitable for CMP systems where degradation due to contention on shared resources may occur. Therefore, we opted to use this definition and employ the *unfairness* metric [16,6], which is defined as follows:

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

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

In our analytical study we assessed the effectiveness of different scheduling algorithms when running several synthetic multi-programmed workloads on an AMP system consisting on two fast cores (FC) and two slow cores (SC). All workloads comprise four single-threaded applications each. In this hypothetical scenario, we assume that applications exhibit fast-to-slow performance ratios that range between 1.2 and 3.4, a similar speedup range than that of the SPEC CPU2006 applications running on the Intel Quick-IA asymmetric system, as reported in [5]. Note that for single-threaded programs, the speedup matches the *speedup factor* (SF) of its single runnable thread, defined as $\frac{IPS_{fast}}{IPS_{slow}}$, where IPS_{fast} and IPS_{slow} are the thread's instructions per second ratios achieved on fast and slow cores respectively. Each row in Table 1 shows the speedup factors (SFs) of the four applications in a specific workload (W_i).

We derived a set of analytical formulas (shown in Table 2) to compute the Unfairness and the Aggregate speedup (ASP) of a workload under a given work-conserving[1] scheduler in this scenario. In deriving the formulas we assume that all applications in the workload run continuously for a certain amount of time T. To make the analytical derivation tractable we also assume that each application exhibits a constant SF during the time interval. Throughout the execution the given scheduler allots each application *app* a certain fast-core time fraction, denoted as F_{app}, such that $0 \leq F_{app} \leq 1$, where $F_{app} = 1$ means that the application would be mapped to a fast core the whole time. Equation 3 makes it possible to obtain the fraction of instructions each application completes on a fast core during the time interval – referred to as f_{app}– based on its speedup factor (SF_{app}) and F_{app}. As evident, the formulas to approximate the ASP and Unfairness only depend on SF_{app} and f_{app}. The detailed derivation process for these formulas as well as for Equation 3 can be found in [17].

$$f_{app} = \frac{1}{\frac{1}{SF_{app}} \cdot \left(\frac{1}{F_{app}} - 1\right) + 1} \tag{3}$$

Figure 1 shows the normalized unfairness and aggregate speedup for the analyzed workloads under five asymmetry-aware schedulers. The first one, denoted as HSP (High-SPeedup), assigns all fast cores to the N_{FC} (number of fast cores) threads in the workload that experience the greatest fast-to-slow speedup (for these applications F_{app}=1); the remaining threads are mapped to slow cores (F_{app}=0). Such a scheduler has been proposed in previous work [13,11]. The second scheduler is an asymmetry-aware round-robin (RR) policy that equally shares fast cores ($F_{app} = \frac{N_{FC}}{n}$) among applications [3,18]. The third scheduler is our proposal, referred to as Prop-SP (Proportional-SPeedup) and explained in detail in Section 3. In the scenario we explored, where workloads consist of equal-priority single-threaded programs, Prop-SP assigns the fast-core share to an application in proportion to its *net speedup* (i.e., $SF_{app} - 1$).

The fourth and fifth schedulers, referred to as Opt-Unfairness and Opt-ASP-Ref, constitute theoretical algorithms. The per-application FC cycle distribution made by Opt-Unfairness ensures the maximum ASP value attainable for the optimal unfairness. Opt-ASP-Ref, on the other hand, achieves the maximum ASP possible ensuring an unfairness value no greater than the one achieved by Prop-SP for a particular workload. We created a simple program which makes use of the analytical formulas in Table 2 to determine per-application fast-core cycle distributions for these theoretical algorithms.

Results from Figure 1 reveal that HSP optimizes the aggregate speedup (the higher ASP, the better) at the expense of obtaining the worst unfairness numbers by far (the higher the unfairness, the worse). As evident, the theoretical Opt-Unfairness scheduler exhibits lower aggregate speedup than HSP in most cases. This fact underscores that, in general, it is not possible to optimize both metrics simultaneously. More importantly, much throughput has to be sacrificed in some cases (up to 20% for W2) to achieve the optimal unfairness. As for the RR scheduler, results highlight that this policy always degrades both fairness and ASP

[1] Such a scheduler does not leave idle cores when the total thread count is greater or equal to the number of cores in the platform.

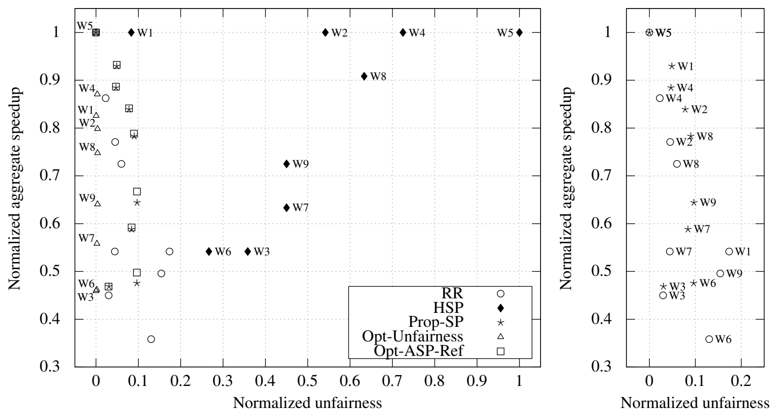

Fig. 1. Aggregate speedup (ASP) and unfairness values for the analyzed workloads under the various schedulers. The closer to the top left corner, the better the ASP-Unfairness tradeoff for the workload in question. Both metrics have been normalized to the (0,1) interval, where 0 represents the minimum value attainable for the metric in the platform and 1 the maximum one. For the sake of clarity, the explicit comparison between RR and Prop-SP has been replicated in a separate figure (right).

compared to Opt-Unfairness, thus providing a suboptimal solution. Notably, RR sacrifices up to 47% of the maximum throughput attainable and in some workloads, such as W1, high throughput reductions are also accompannied by fairness degradation. Finally, the results showcase good properties regarding the Prop-SP scheduler. First, it delivers higher aggregate speedup than Opt-Unfairness and RR across the board. Second, despite the slight fairness degradation, Prop-SP ensures unfairness numbers within 0-10% of the maximum attainable for all workloads (clearly, this is not always the case for HSP and RR). Third, results of the theoretical Opt-ASP-Ref scheduler reveal that Prop-SP delivers ASP numbers very close to the maximum attainable for the provided unfairness.

3 The Prop-SP Scheduler

3.1 The Algorithm

Prop-SP assigns threads to fast and slow cores so as to preserve load balance in the AMP, and periodically migrates threads between fast and slow cores to ensure that they run on fast cores for a specific amount of time. To perform thread-to-core assignments, it relies on two mechanisms: *fast-core credit allocation* and *inter-core swaps.*

Fast-Core Credit Allocation is a mechanism to control the amount of fast-core cycles allotted to the running threads on an AMP. At a high level, fast-core credit allocation works as follows. Each thread has a fast-core credit counter associated with it. When a thread runs on a fast core it consumes credits. Threads that have fast-core credits left (i.e., their credit counter is greater than zero) are preferentially assigned to fast cores by Prop-SP. Every so often, the OS triggers a credit assignment process that allots fast-core credits to applications with runnable threads. The time period elapsed between two consecutive system-wide credit assignments is set dynamically by the scheduler. We will refer to this elapsed period as the *execution period.* Note that we borrowed the idea of associating credits to threads from Xen's Credit Scheduler (CS) [4]. However, credit distribution in Prop-SP is completely different from that of CS.

Prop-SP awards fast-core credits to each application based on its associated *dynamic weight*, which is defined as the product of its net speedup (speedup minus one) and its *static weight.* In this context, the speedup indicates the relative benefit that the application would derive if all fast cores in the AMP were devoted to running threads from this application, with respect to running all threads on slow cores. The speedup is estimated at runtime by Prop-SP without the user intervention (see Section 3.2). The *static weight*, by contrast, is derived directly from the application priority (set by the user).

The credit assignment process entails three steps as detailed in Algorithm 1. After computing dynamic weights (step 1), Prop-SP allots credits to each application based of its dynamic weight in competition with the sum of the dynamic weights of all applications (step 2). Because the actual length of the next execution period is computed afterwards so as to control the migration rate (we will elaborate on this issue later), the credit distribution performed in step 2 is done assuming a fixed-width reference execution period. Once the length of the execution period has been determined, awarded per-application credits are scaled to the actual interval length. Finally, credits awarded to the application are then distributed among its runnable threads (step 3). For sequential programs, per-thread credit-distribution entails increasing the credit counter of the only thread by the amount of credits awarded. For multi-threaded applications, Prop-SP supports two per-thread credit distribution schemes: Even and BusyFCs. Even distributes credits uniformly across runnable threads in the application. BusyFCs goes sequentially through runnable threads and assigns each one the maximum amount of credits it can consume in the next execution period (`cred_per_fc_next_period`) until there are no more credits left to share. We found that the Even scheme is well-suited to coarse-grained parallel applications while BusyFCs turns out beneficial for fine and mid-grained parallel programs. The associated experimental analysis has been omitted due to space constraints.

Inter-Core Swaps is a thread-migration mechanism that ensures that threads with fast-core credits get a chance to use up their credits without disturbing load balance. In order to illustrate how this mechanism works, let us consider an AMP with one fast core and one slow core. Suppose that there are two threads with fast-core credits running on the system, each one mapped to a different core to preserve load balance. Eventually, the thread running on the fast core runs out of fast-core credits. At this point, the scheduler *swaps* both threads between cores to make sure the thread that was running on the slow core gets a chance to consume its fast-core credits while maintaining load balance.

Algorithm 1: Credit Assignment Algorithm

```
{ • *R is the set of applications with runnable threads.*
  • $N_{FC}$ *is the number of fast cores (FCs).*
  • CRED_1FC_REF *is the amount of credits consumed on each FC during an
  execution period used as reference.*
  • cred_per_fc_next_period *is the amount of credits consumed on each FC during
  the next execution period.* }
S:= [ ];  total_weight:=0;  total_credits:=CRED_1FC_REF $* N_{FC}$;
{ STEP 1 ⇒ Compute apps' dynamic weight and total_weight }
foreach app in R do
    $\text{speedup}_{app}$:= estimate speedup for app;
    $\text{dyn\_weight}_{app} := (\text{speedup}_{app} - 1) * \text{static\_weight}_{app}$;
    $\text{total\_weight} := \text{total\_weight} + \text{dyn\_weight}_{app}$;
    Insert app into S so as to keep S sorted in descending order by $\text{dyn\_weight}_{app}$;
end
{ STEP 2 ⇒ Assign credits to apps based on $\text{dyn\_weight}_{app}$ }
foreach app in S do
    $\text{credit}_{app} := \frac{\text{total\_credits} * \text{dyn\_weight}_{app}}{\text{total\_weight}}$;
end
{ STEP 3 ⇒ Determine the length of the next execution period
and distribute credits among threads }
Compute cred_per_fc_next_period;
scale_factor:=cred_per_fc_next_period/CRED_1FC_REF;
foreach app in S do
    $\text{credit}_{app} := \text{credit}_{app} * \text{scale\_factor}$;
    Distribute $\text{credit}_{app}$ credits among threads in app
end
```

3.2 Determining the Speedup

At runtime, Prop-SP needs to obtain the relative speedup that an application derives from using all fast cores in the AMP. This value is used by the credit distribution algorithm to compute the application's dynamic weight.

As mentioned in Section 2, the speedup of a single-threaded application matches the SF of its single runnable thread. To determine a thread's SF online, Prop-SP feeds a platform-specific estimation model with values from diverse performance metrics collected over time[2] (such as the IPC or the last-level-cache miss rate). In this work, we leverage the technique proposed in our previous work [19] to aid in the construction of SF estimation models. This technique, which has been proven successful in a AMP prototype system where cores differ in microarchitecture, enables to generate SF models by analyzing offline-

[2] In our setting, performance counters are sampled every 200ms, which leads to negligible overhead associated with sampling and SFs estimation.

collected performance counter data from a representative set of single-threaded CPU-bound programs.[3]

To obtain a speedup estimate for a multithreaded application, several factors in addition to the SF must be taken into account [9,19], such as its amount of thread-level parallelism (TLP) or how fast-core credits are distributed among its threads. Prop-SP makes use of the following equations to estimate the application speedup under the BusyFCs and the Even credit-distribution schemes:

$$SP_{BusyFCs} = \frac{SF-1}{(\lfloor \frac{N-1}{N_{FC}} \rfloor + 1)^2} + 1 \qquad SP_{Even} = \frac{MIN(N_{FC}, N)}{N} \cdot (SF-1) + 1$$

where N is the number of threads in the application, N_{FC} is the number of fast cores in the AMP and SF is the average speedup factor of the application threads. The detailed derivation process for these formulas can be found in our previous work [19,17].

4 Experimental Evaluation

In our experiments, we analyzed two variants of Prop-SP (static and dynamic), which follow different approaches to determine a thread's SF. The *base* implementation of Prop-SP, referred to as *Prop-SP (dynamic)*, estimates SFs online using hardware counters. *Prop-SP (static)*, on the other hand, asummes a constant SF value for each thread, measured prior to the execution. We compare both versions of Prop-SP against four previously-proposed schemes: RR [3,18], A-DWRR [15], CAMP [19] and HSP (High-SPeedup) [3,11]. In previous work [18], we observed that considering the speedup of the application as a whole rather than the speedup of individual threads when making thread-to-core mappings leads to higher throughput in scenarios where parallel applications are present. As such, for a fairer comparison, we modified HSP to perform thread-to-core assignments taking into account the application-wide speedup rather than per-thread speedup factors.

All the evaluated algorithms have been implemented in the Solaris kernel and tested on real multicore hardware made asymmetric by reducing the processor frequency of a subset of cores in the platform. In particular, we used a multicore server consisting of two AMD Opteron 2435 "Istanbul" hex-core processors (12 cores). Each chip includes a 6MB shared L3 cache shared among cores. Emulated AMP configurations on this system consist of "fast" cores that operate at 2.6GHz and "slow" cores running at 800MHz. To evaluate the different scheduling algorithms, we used two AMP configurations: (1) 2FC-2SC – including two chips with one fast core and one slow core (2) 2FC-10SC – two chips with one fast core and 5 slow cores each.

[3] In this work we obtained the SF estimation models by analyzing offline-collected data from a subset of the SPEC CPU 2006 benchmarks. Note that we also experimented with applications different to those employed to generate the models.

Table 3. Multi-application workloads consisting of single- and multithreaded programs

Categories	Benchmarks	Categories	Benchmarks
3STH-1HPH	hmmer, gobmk, h264ref, fma3d_m(9)	4STH	povray, gobmk, bzip2, sjeng
3STH-1HPL	povray, gamess, gobmk, swim_m(9)	3STH-1STM	povray, h264ref, perlbench, astar
2STH-1PSH-1HPM	gamess, bzip2, BLAST(4), wupwise_m(6)	3STH-1STL_A	hmmer, namd, perlbench, soplex
1STH-1STM -1STL-1PSH	gamess, astar, soplex, blackscholes(9)	3STH-1STL_B	hmmer, h264ref, gobmk, milc
1PSH-1PSL	semphy(6), FFTW3D(6)	2STH-2STM_A	povray, bzip2, leslie3d, sphinx3
2PSH-1HPM	BLAST(4), semphy(4), wupwise_m(4)	2STH-2STM_B	gamess, gobmk, xalancbmk, astar
1PSH-1HPL	semphy(6), equake_m(6)	2STH-2STL_A	hmmer, gobmk, lbm, soplex
1HPH-1HPL	fma3d_m(6), equake_m(6)	2STH-2STL_B	povray, h264ref, lbm, omnetpp
1PSH-1HPH	blackscholes(6), fma3d_m(6)	1STH-1STM-2STL	sjeng, leslie3d, lbm, soplex

Our evaluation targets multi-application workloads consisting of HPC benchmarks from diverse suites (SPEC CPU 2006 and OMP 2001, PARSEC, NAS Parallel Benchmarks and Minebench). We also experimented with BLAST – a bioinformatics benchmark – and FFTW3D – an HPC benchmark performing the fast Fourier transform. In all experiments, the sum of the number of threads of all applications was set to match the number of cores in the platform, since this is how runtime systems typically configure the number of threads for CPU-bound workloads like the ones we used. We ensure that all applications in the workload are started simultaneously and when an application terminates it is restarted repeatedly until the longest application in the set completes three times. For each application in a workload, CT_{sched} is calculated as the geometric mean of its completion times for the various executions. We measure CT_{fast} for an application by tracking its completion time when running alone in the AMP with its best-performing per-thread credit distribution scheme.

Table 3 shows the analyzed multi-application workloads. The first nine workloads (left) consist of both sequential and parallel applications; the last nine (right) comprise sets of single-threaded programs. In creating the workloads, we categorized applications into three groups with respect to their parallelism: highly parallel (HP), partially sequential (PS) –parallel applications with a sequential component of over 25% of the total execution time– and single-threaded (ST). In order to cater to applications' SFs as well, we further divided the three aforementioned application groups into three subclasses based on their SFs – high (H), medium (M) and low (L). The application categories are shown in the table in the same order as the corresponding benchmarks. For example, in the 1PSH-1HPL category, semphy is the PSH application and equake_m is the HPL one. The number in parentheses next to the name of each multithreaded application indicates the number of threads it runs with.

Figure 2 shows the aggregate speedup and unfairness for the workloads under the various schedulers. Overall, HSP and CAMP, which assign high-speedup applications to fast cores, yield the highest system throughput in most cases but fail to deliver fairness accross the board. RR and A-DWRR, on the other hand, do rather a good job in terms of both fairness and throughput for workloads including single-threaded applications only. However, when multithreaded pro-

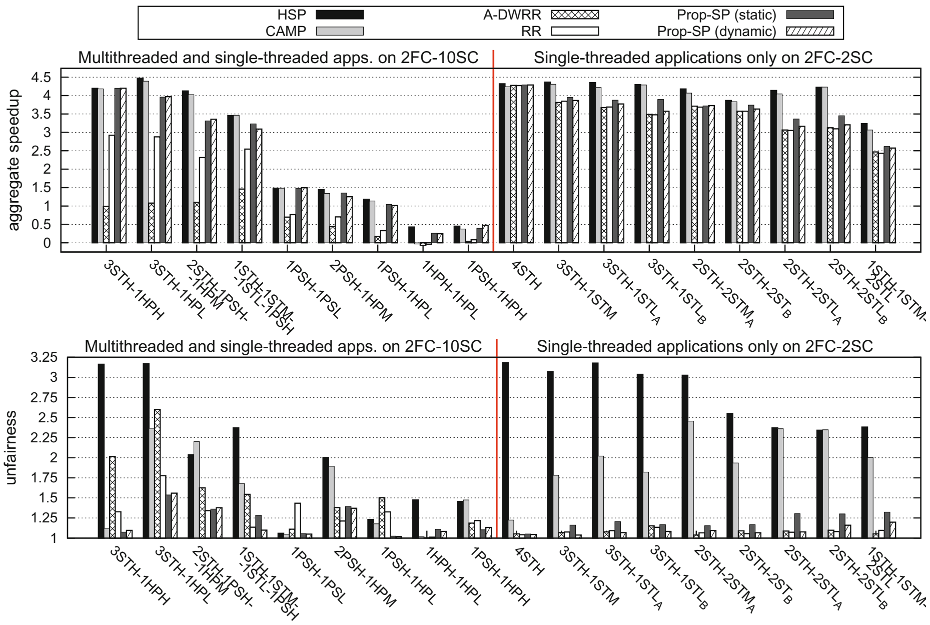

Fig. 2. Aggregate speedup and unfairness of the investigated scheduling algorithms

grams are present in the workload, both schedulers degrade the system throughput significantly. In this scenario, A-DWRR awards higher fast-core share to those applications with a higher thread count. As shown in [18], this may lead to throughput degradation since applications with a high active thread count may experience low benefit from using the scarce fast cores in the platform.

The results reveal that Prop-SP is able to make efficient use of the AMP and improve the throughput-fairness tradeoff for a wider range of workloads. Overall, these benefits are especially pronounced for workloads including multithreaded programs. In this scenario, Prop-SP is able to match the performance of HSP and CAMP for 3STH-1HPH, 1PSH-1PSL and 1PSH-1HPH, while performing in a close range for the remaining application mixes. At the same time, it achieves much lower unfairness numbers than HSP and CAMP across the board and exhibits comparable unfairness to A-DWRR and RR. Moreover, we observe that the inacuracies of the SF estimation model used by Prop-SP (dynamic), do not prevent it from reaping benefits similar to those of the static version.

5 Related Work

A large body of work has advocated the benefits of AMPs over symmetric CMPs [13,12,9]. Despite these benefits, AMP systems pose significant challenges to the system software [15]. OS scheduling is one of the most critical challenges, and this is the focus of our paper.

Most existing asymmetry-aware schedulers strive to optimize the system throughput. Schedulers targeting workloads consisting of single-threaded programs only [13,3,11,21,18] aim to maximize throughput by running on fast cores those applications with a higher SF. To maximize throughput in workload scenarios including multithreaded programs, the amount of thread-level parallelism (TLP) in the applications must be taken into account. In this scenario, some schedulers make use of fast cores in the AMP as accelerators for serial execution phases in parallel applications [18,19,10]. These schemes, however, do not attempt to ensure fairness. In our proposal, the OS-level scheduler acts as a global arbiter that delivers fairness by adjusting the fast-core share allotted to the various programs in multiapplication scenarios.

To the best of our knowledge, A-DWRR [15] is the first scheduler aiming to deliver both fairness and priority enforcement on asymmetric single-ISA multicore systems. Unlike Prop-SP, A-DWRR does not take into account that applications derive different speedups when using fast cores in the platform and that these speedups may vary over time. Moreover, A-DWRR performs CPU-time allocation on a per-thread basis rather than on a per-application basis. As our experimental results reveal, these factors may lead A-DWRR to degrading the system throughput significantly and prevent this scheduler from ensuring an even slowdown for equal-priority applications on AMPs, especially when multithreaded applications are present in the workload.

6 Conclusions

In this paper we proposed Prop-SP, a scheduler that aims to improve the balance between fairness and throughtput on asymmetric multicores. To make this possible, Prop-SP exploits the diversity in the fast-core efficiency of a workload to even out the slowdown experienced by simultaneously running applications (based on their priorities) when sharing the fast cores of an AMP. We implemented Prop-SP in the Solaris kernel and compared it against several state-of-the-art asymmetry-aware schedulers. Our experiments reveal that Prop-SP is able to make efficient use of the AMPs and improve the throughput-fairness tradeoff for a wider range of workloads. The benefits of the Prop-SP policy are especially pronounced for workloads including multithreaded programs.

Key elements for the success of Prop-SP are the credit-based mechanism enabling the scheduler to adjust the fast-core share allotted to the different programs and its reliance on estimation models to approximate application speedup online. As shown in previous work [19], asymmetry-aware schedulers relying on SF estimation models, such as Prop-SP, can be seamlessly extended to different forms of performance asymmetry. Evaluating Prop-SP on cutting-edge AMP prototypes [5] is an interesting avenue for future work.

Acknowledgements. This work has been supported by the Spanish government through the research contract TIN2012-32180 and the HIPEAC^3 European Network of Excellence.

References

1. Alameldeen, A.R., Wood, D.A.: IPC considered harmful for multiprocessor workloads. IEEE Micro 26(4) (2006)
2. ARM: Benefits of the big.LITTLE Architecture (2012)
3. Becchi, M., Crowley, P.: Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In: Proc. of CF 2006, pp. 29–40 (2006)
4. Cherkasova, L., Gupta, D., Vahdat, A.: Comparison of the three CPU schedulers in Xen. SIGMETRICS Perform. Eval. Rev. 35(2), 42–51 (2007)
5. Chitlur, N., et al.: QuickIA: Exploring heterogeneous architectures on real prototypes. In: Proc. of HPCA 2012, pp. 1–8 (2012)
6. Ebrahimi, E., et al.: Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In: ASPLOS 2010 (2010)
7. Eyerman, S., Eeckhout, L.: System-level performance metrics for multiprogram workloads. IEEE Micro 28(3) (2008)
8. Gabor, R., Weiss, S., Mendelson, A.: Fairness and throughput in switch on event multithreading. In: Proc. of MICRO 2006 (2006)
9. Hill, M.D., Marty, M.R.: Amdahl's Law in the Multicore Era. IEEE Computer 41(7), 33–38 (2008)
10. Joao, J.A., et al.: Utility-based acceleration of multithreaded applications on asymmetric CMPs. In: Proc. of ISCA 2013, pp. 154–165 (2013)
11. Koufaty, D., Reddy, D., Hahn, S.: Bias Scheduling in Heterogeneous Multi-core Architectures. In: Proc. of Eurosys 2010 (2010)
12. Kumar, R., et al.: Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In: Proc. of MICRO, vol. 36 (2003)
13. Kumar, R., et al.: Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In: Proc. of ISCA 2004 (2004)
14. Kwon, Y., et al.: Virtualizing performance asymmetric multi-core systems. In: Proceedings of ISCA 2011 (2011)
15. Li, T., et al.: Operating system support for overlapping-ISA heterogeneous multicore architectures. In: HPCA 2010, pp. 1–12 (2010)
16. Mutlu, O., Moscibroda, T.: Stall-time fair memory access scheduling for chip multiprocessors. In: Proc. of MICRO 2007 (2007)
17. Pousa, A., et al.: Theoretical study on the performance of an asymmetry-aware round-robin scheduler. TR - 5028A. Dept. of Computer Architecture. UCM (2012), `https://artecs.dacya.ucm.es/sites/default/files/dacya-tr5028A.pdf`
18. Saez, J.C., et al.: A Comprehensive Scheduler for Asymmetric Multicore Systems. In: Proc. of ACM Eurosys 2010 (2010)
19. Saez, J.C., et al.: Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. ACM TOCS 30(2) (April 2012)
20. Saez, J.C., et al.: Delivering fairness and priority enforcement on asymmetric multicore systems via OS scheduling. In: Proc. of ACM SIGMETRICS (2013)
21. Shelepov, D., et al.: HASS: A Scheduler for Heterogeneous Multicore Systems. ACM SIGOPS OSR 43(2) (2009)