Exploring the Design Space of Fair Scheduling Supports for Asymmetric Multicore Systems

Changdae Kim and Jaehyuk Huh, *Member, IEEE*

Abstract—Although traditional CPU scheduling efficiently utilizes multiple cores with equal computing capacity, the advent of multicores with diverse capabilities pose challenges to CPU scheduling. For such asymmetric multi-core systems, scheduling is essential to exploit the efficiency of core asymmetry, by matching each application with the best core type. However, in addition to the efficiency, an important aspect of CPU scheduling is fairness in CPU provisioning. Such uneven core capability is inherently unfair to threads and causes performance variance, as applications running on fast cores receive higher capability than applications on slow cores. Depending on co-running applications and scheduling decisions, the performance of an application may vary significantly. This study investigates the fairness problem in asymmetric multi-cores, and explores the design space of OS schedulers supporting multiple fairness constraints. In this paper, we consider two fairness-oriented constraints, *minimum fairness* for the minimum guaranteed performance and *uniformity* for performance variation reduction. This study proposes four scheduling policies which guarantee a minimum performance bound while improving the overall throughput and reducing performance variation too. The proposed fairness-oriented schedulers are implemented for the Linux kernel with an online application monitoring technique. Using an emulated asymmetric multi-core with frequency scaling and a real asymmetric multi-core with the big.LITTLE architecture, the paper shows that the proposed schedulers can effectively support the specified fairness while improving overall system throughput.

Index Terms—Fair scheduling, asymmetric multicore, performance variance

1 Introduction

Traditional CPU scheduling by the operating system efficiently utilizes multiple cores with the same computing capability. However, recent architectural changes pose challenges for the CPU scheduling with the advent of cores with different computing capabilities in a system. One example of such architectural changes is the asymmetric multi-core processor (AMP) with multiple types of cores, supporting the same instruction-set architecture (ISA) with different computing capabilities [1], [2], [3]. Furthermore, process variation incurs different maximum frequencies for cores in a multi-core [4], [5], and common dynamic voltage and frequency scaling (DVFS) also allows a CPU to have cores with different settings for computing capability and energy consumption.

To fully exploit the potential of such asymmetric multi-cores, scheduler support is crucial. While scheduling for asymmetric multi-cores has been widely studied [1], [4], [6], [7], [8], [9], [10], [11], [12], [13], most of the studies aim at maximizing overall throughput by exploiting asymmetry and application behaviors. Such throughput-maximizing scheduling assigns fast cores to applications with high relative performance gains with fast cores compared to slow cores.

• *The authors are with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon 305-701, Republic of Korea. E-mail: {cdkim, jhuh}@calab.kaist.ac.kr.*

Manuscript received 31 Mar. 2017; revised 10 Jan. 2018; accepted 14 Jan. 2018. Date of publication 22 Jan. 2018; date of current version 7 July 2018. (Corresponding author: Jaehyuk Huh.)
Recommended for acceptance by G. Min.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2796077

However, an important but neglected aspect of CPU scheduling in the prior studies is fairness of CPU provisioning. As we will show in Section 4, throughput-maximizing scheduling often sacrifices the fairness much more than its benefit on throughput. Furthermore, such fairness has become critical as recent cloud computing environments are required to provide consistent performance for their guest machines in consolidated systems. Although there have been several studies to improve fairness for asymmetric multi-cores [7], [8], [11], [13], the schedulers do not support minimum performance guarantee, which is essential for such consolidated systems.

In this paper, we explore two different aspects of fairness. The first one is to guarantee a minimum performance regardless of uneven core capability. Such minimum fairness guarantee sets the lower bound of performance for each application. The second aspect is to reduce relative performance variance. For each application, fair scheduling must reduce the variation of performance degradation normalized to an ideal isolated run. These aspects support two different goals of fair scheduling, first, setting a certain limit in possible performance degradation by asymmetry in core capability, and second, reducing performance variation. Furthermore, while aiming the two fairness-oriented goals, the overall throughput must be improved to exploit the performance/energy efficiency from asymmetric cores. Prior throughput-maximizing schedulers often sacrifice fairness of CPU provisioning excessively to gain only a small amount of extra throughput.

To investigate how fairness should be supported in asymmetric multi-cores, this paper explores the design space of fairness-oriented schedulers for asymmetric

multi-cores, which allow a certain level of fairness to be guaranteed while improving throughput. The first scheduler, `min-fair`, always supports a fixed level of minimum fairness constraint, guaranteeing that the performance of no application is degraded beyond a preset limit compared to the fair CPU allocation. The second scheduler, `uniformity-fair`, supports a fixed level of performance variation limit, providing the performance variation is within a preset bound. The third scheduler, `sim-fair`, opportunistically reduces performance variation of the prior throughput-maximizing scheduler by relaxing the strict throughput-oriented allocation. The final scheduler, `combined-fair`, combines the aforementioned three schedulers. The fairness-oriented schedulers provide the system administrator with the mechanisms to choose different ways of setting fairness requirements. All the schedulers still attempt to improve the overall throughput as long as fairness constraints are satisfied.

To show such fairness-oriented schedulers are feasible, we modified the CFS scheduler in Linux 3.10.96 to support fine-grained scheduling for different core capabilities. We implemented the scheduler to work effectively for two different core capabilities with dynamic voltage frequency scaling. A challenge in its implementation is the estimation of fast core speedup. In the prior work, the performance gain with fast cores, *fast core speedup*, is estimated online or offline indirectly. To improve the accurate estimation with low overheads, we have implemented an exploration-based fast core speedup estimation.

We evaluated our schedulers on two different setups with uneven core capability. The first setup is an emulated asymmetric multi-core processor using DVFS to mimic core asymmetry. The second one uses a real asymmetric multi-core processor with the ARM big.LITTLE architecture [3]. The results with various mixes show that our schedulers guarantee the specified fairness and still improve the overall throughput.

The remainder of the paper is organized as follows. Section 2 defines fairness in asymmetric multi-cores, and Section 3 presents the prior work. Section 4 quantitatively analyzes the fairness problem of the prior throughput-maximizing scheduler. In Section 5, we propose the fairness-oriented scheduling policies. Section 6 describes the implementation issues including the fast core speedup estimation mechanism. The experimental results on real machines are shown in Section 7, and Section 8 concludes this paper.

2 Fairness for Asymmetric Multi-Cores

This section discusses the fairness aspects of scheduling on asymmetric multi-cores. First, the definition of fair scheduling is discussed. Unlike symmetric multi-cores, which the amount of CPU cycles solely affects the application performance in the perspective of scheduling, on asymmetric multi-cores, the type of CPU cycles also affects the application performance. Thus, the definition of fair scheduling should be refined. Second, we also discuss *base performance* which is required to define fair scheduling on asymmetric multi-cores. Third, we discuss how to achieve such fair scheduling state by adjusting the amount of CPU cycles for each CPU types. Last, we discuss fairness metrics to evaluate the scheduling on asymmetric multi-cores.

Since asymmetric multi-cores have advantages on the power or area efficiency [1], most of the prior studies have been focused on the throughput aspect of scheduling, and proposed throughput-maximizing schedulers which we call `max-perf` in the remainder of this paper. However, we still need to consider fairness issues due to the following reasons. First, exploiting efficiency from asymmetry is not always beneficial enough. As we will show in Section 4, `max-perf` policy often sacrifices the fairness much more than its benefit on throughput. Second, even though some systems adopt asymmetric multi-cores due to its efficiency, the fairness among threads in the system still may be the most important issue. For example, cloud providers can adopt asymmetric multi-cores to reduce power cost, but they always need to satisfy service level agreements (SLA) on application performance for users. In popular cloud services such as EC2, the SLA for processors is represented with a normalized performance unit. The underlying guarantee is to provide consistent performance for the promised performance unit regardless of co-runners.

Through this paper, we consider the following environment. First, our scheduler manages a physical system. It does not span multiple physical systems. Second, to simplify analysis and discussion, we consider only two types of cores, *fast core* and *slow core*. Third, we assume that each thread belongs to different applications. Note that the most schedulers manage threads independently. We will discuss how to support the multi-threaded applications in Section 7.6. Finally, we focus on non-real time applications. For real time applications, meeting deadline is more important than fairness. Also, as Linux scheduler which has a modular scheduler design, we can use different schedulers for real time applications.

2.1 Definition of Fair Scheduling on AMP

On symmetric multi-core processors (SMP), there is one knob for schedulers, the amount of CPU share. Users set the weight of threads (w^i), and schedulers adjust the amount of CPU share (c^i) of each thread. The fair scheduling on SMP is defined as the amount of CPU share is proportional to the weight of thread [14], [15]. Let N be the number of threads and C be the number of CPUs in the system

$$c^i = \frac{w^i}{\sum_{j=1}^{N} w^j} C.$$

On asymmetric multi-core processors, another knob is added, the type of CPU share. Schedulers adjust the amount of fast core share (f^i) and slow core share (s^i) of each thread, and these values affect the performance of the thread. Note that $f^i + s^i = c^i$ by definition. The performance depends on the characteristic of the thread, *fast core efficiency* (e^i). This represents the relative performance on a fast core compared to that on a slow core. When the performance of an application ($perf^i$) is represented as the inverse of the execution time ($exectime^i$), *fast core efficiency* is defined as follows:

$$e^i = \frac{perf^i_{fast}}{perf^i_{slow}} = \frac{exectime^i_{slow}}{exectime^i_{fast}},$$

where the subscript *fast* or *slow* means that the value is extracted from the case that the application runs entirely on a fast or a slow core.

Since the characteristic of a thread as well as its fast core share and slow core share affect the performance of the thread, it is necessary to set a performance baseline to define a fair scheduling on AMP. We call the baseline as *base performance*. For example, Van Craeynest et al. [13] used a performance when an application always runs on a fast core in an isolated environment for their base performance. Section 2.2 discusses *base performance* in detail.

Finally, we use the following definition for the fair scheduling state: *a scheduling on AMP system is fair, if the amount of CPU share of all threads are proportional to their weights and all threads experience same slowdown or speedup from the base performance*. Then, we define `max-fair` policy as a scheduling policy which achieves the fair scheduling state. How to implement `max-fair` policy with the given *base performance* will be explained in Section 2.3.

2.2 Base Performance

As discussed in the previous section, to define a fair scheduling state, the baseline performance for each application is necessary, as the minimum fairness and uniformity metrics use normalized performance against the base performance.

The performance with fast core or slow core has been used as the baseline performance in the prior work [7], [13]. *Fast only* base assumes that all applications run on fast cores, and takes their performance as the baseline performance. Similarly, *slow only* base takes the performance when all applications run on slow cores as the baseline. The base performances have the absolute values and do not depend on core configurations and the number of threads in a system. Also, users can easily guess the performance if they know what the baseline core is. Thus, using these base performances can facilitate the service-level agreement support. However, with *fast only* or *slow only*, the maximum achievable value of minimum fairness depends on the combination of applications in a system. For example, depending on the corunning applications, 70 percent minimum fairness with *fast only* base performance may not be achievable. In the prior work, [13] targeted only uniformity as a fairness metric and does not have such problems.

An alternative is *fair share* base performance, proposed by Kwon et al. [11]. It is defined as the performance of applications when an equal core share for fast and slow cores are assigned to all applications. In other words, all applications have the same chance to use fast cores. It provides the exact upper bound of minimum fairness regardless of the characteristics of running applications. The maximum value of minimum fairness is always 100 percent, and it can be achieved by assigning core share as its definition. However, *fair share* can be less attractive from the perspective of supporting the service-level agreement on the application performance. The number of active threads and the number of cores for each type affect the base performance, as how much fast core share can be assigned to an application is dependent upon those numbers.

Therefore, in this paper, we explore three types of base performances described so far. Cloud providers may use *fast only* or *slow only* base performance to let users can easily guess the provided performance while hiding the detailed information about the system. The problem with changes in the maximum achievable minimum fairness can be solved by other ways. On the other hand, normal users who own systems with asymmetric multicores may use *fair share* base performance, if they want an exact control of their own systems regardless of characteristics of running applications.

TABLE 1
Variables and Meanings

e^i	fast core efficiency of thread i	N	the number of threads
f^i	fast core share of thread i	F	the number of fast cores
s^i	slow core share of thread i	S	the number of slow cores
c^i	total core share of thread i	C	the number of all cores

Another issue in defining base performance is whether the shared resource effect should be removed or not. Shared resources such as shared last-level caches cause the performance interference across applications on different cores. The base performance by Craeynest et al. [13] excludes the shared resource effect, while the base performance by Kwon et al. [11] includes the performance interference.

In this paper, we decide not to consider the shared resource effect for the base performance due to the following reasons. First, we want to make a feasible scheduler on the currently available systems. Estimating the performance without the shared resource effect often requires the extra hardware support or large sampling overhead. Second, the interference from the shared resources can be solved at the shared resources themselves, and is far from the major role of CPU schedulers. Third, in most cases, the performance gap between fast and slow cores have a primary impact on application performance. Although our scheduler designs do not consider the performance interferences, all experimental results on real machines include the effect.

2.3 Achieving Fair Scheduling State

In this section, we describe how to achieve the fair scheduling state for each base performance definition. Table 1 summarizes the variables we use in this section. Also, we assume that total core share of all threads are allocated proportionally to their weight. The outcome of this analysis is to find the fast core share for each thread which equalize the normalized performance compared to the base performance. For the slow core share, $s^i = c^i - f^i$ by definition.

For *fair share* base, the base performance defines the amount of fast core share for the fair scheduling. To maintain the proportional fairness for total core share, the amount of fast core share of thread i should be as follows:

$$f^i = \frac{c^i}{\sum_{j=1}^{N} c^j} F.$$

For *slow only* base, the definition of fair scheduling on AMP implies the following conditions:

$$\forall i, j, \frac{e^i f^i + s^i}{c^i} = \frac{e^j f^j + s^i}{c^j} \text{ and } \sum_{i=1}^{N} f^i = F.$$

The left condition represents that the speedup due to the fast core share normalized to the slow core only

performance should be equal, and the right condition indicates that the total fast core share should be equal to the total number of fast cores. Note that when we say the slow core only performance as 1, the performance with given f_i and s_i can be presented as $e_i f_i + s_i$. Then, since the slow core only performance is the performance when the application consumes all of its core share on the slow cores, the slow core only performance can be presented as c^i.

Using $s^i = c^i - f^i$, the left condition can be rearranged as $f^i = ((e^j - 1)f^j + (c^j - c^i))/(e^i - 1)$. By applying this to the right condition, we can get the solution for f^j

$$f^j = \frac{\frac{1}{e^j - 1}}{\sum_{i=1}^{N} \frac{1}{e^i - 1}} \left(F - \sum_{i=1}^{N} \frac{c^j - c^i}{e^i - 1} \right).$$

Note that $e^j - 1$ is the additional performance ratio when a thread receives fast core share instead of slow core share. Then, $1/(e^j - 1)$ is the fast core share ratio to equalize the additional performance. The second term in the parenthesis reflects the proportional fairness in total core share with the consideration of the fast core efficiency of applications.

For *fast only* base, we should consider the following conditions:

$$\forall i, j, \frac{e^i f^i + s^i}{e^i c^i} = \frac{e^j f^j + s^j}{e^j c^j} \text{ and } \sum_{i=1}^{N} f^i = F.$$

The conditions are similar to *slow only* base, but the denominator in the left condition is $e^i c^i$, which represents the performance when the application consumes its total core share on fast cores.

We can use the similar procedure with *slow only* base to get the final solution. The result is as follows:

$$f^j = \frac{\frac{e^j c^j}{e^j - 1}}{\sum_{i=1}^{N} \frac{e^i c^i}{e^i - 1}} F - \frac{1}{e^j} \left(\frac{e^j c^j}{e^j - 1} \right) + \frac{\frac{e^j c^j}{e^j - 1}}{\sum_{i=1}^{N} \frac{e^i c^i}{e^i - 1}} \sum_{i=1}^{N} \frac{1}{e^i} \left(\frac{e^i c^i}{e^i - 1} \right).$$

The first term of the solution is similar to *slow only* base case, but the numerator here is $e^i c^j$. It means that the additional performance is normalized to the performance on fast cores. The second and third terms are needed since the performance from slow core share is not directly proportional to slow core share. It also depends on the fast core efficiency with fast only base. The second term substracts the performance interchangeable with slow core share as assuming that threads receive fast core share by the first term, and the third term re-distributes the summation of the interchangeable performance to equalize the additional performance from fast core share.

Unfortunately, the solutions for *fast only* and *slow only* base performance might give infeasible values, such as $f^i < 0$ or $f^i > 1$. This is similar to *infeasible weight* problem [15], [16] on SMP scheduling. Depending on the distribution of the fast core efficiency, it may be impossible to make a fair scheduling. For such cases, we make $f_i = 0$ if $f_i < 0$ in the solution, and $f_i = 1$ if $f_i > 1$ in the solution. Then, we exclude the exceptional threads with the core share they receive, and calculate the solution again.

2.4 Fairness Metrics

To evaluate our scheduling policies, we use the following metrics. We define T^i, the *throughput* of an application i as the performance normalized to the *base performance* ($perf_{base}$). With our definition of the fair scheduling on AMP, if a scheduling is fair, *throughputs* of all applications should be same. For the *system-wide throughput* metric, T, we use the arithmetic mean as follows:

$$T^i = \frac{perf^i}{perf^i_{base}} \qquad T = \frac{1}{n} \sum T^i.$$

With this metric, we define `max-perf` scheduling which maximizes the system-wide throughput without considering fairness. Suppose the number of fast cores is F. Then, `max-perf` selects the F applications with the highest fast core efficiency, and schedules them on the fast cores. The rest of applications are scheduled to the slow cores.

For fairness, we use two different metrics, `minimum fairness (minF)` and `uniformity`. Minimum fairness mandates the limit of maximum performance degradation compared to the base performance of each application. Uniformity is how uniform the performances of applications are relatively to the base performances respectively. Minimum fairness, the minimum performance relative to the base performance, is defined as follows. Note that T^i is the performance normalized to the base performance

$$minF = \min(T^1, T^2, \ldots, T^N),$$

where N is the number of applications. The range of `minimum fairness` value is affected by base performance.

Second, the uniformity metric is the fairness metric proposed by Van Craeynest et al. [13]. It is based on a standard deviation of normalized performance of each application. Uniformity is defined as follows:

$$Uniformity = 1 - (\sigma_{T^i}/\mu_{T^i}),$$

where σ_{T_i} is the standard deviation of application throughputs and μ_{T_i} is the average of application throughputs. For the normalization point, the original study [13] uses an estimated isolated performance on fast cores. In this paper, we use the more general form, the base performance.

3 Prior Work

Kumar et al. proposed an asymmetric multi-core processor design and showed its potential to improve area and energy efficiency [1], [17]. Recently, AMPs have been realized in academic and commercial designs such as FabScalar project [2] and big.LITTLE architecture [3]. Furthermore, process variation incurs different maximum frequencies for cores in a system and results in unintended asymmetric multi-cores [4], [5]. Scheduling mechanisms for asymmetric multi-cores have been studied to exploit the potential of it. In the rest of this section, we will discuss prior fairness aware schedulers for asymmetric multi-cores in detail, then summarize the other work.

3.1 Prior Fairness-Aware Schedulers

There have been some studies to incorporate fairness into the scheduling problem of asymmetric multi-cores. First,

TABLE 2
Comparison with Prior Approaches

	scaled load balancing [7]	R%-fair [11]	guaranteed-fairness [13]	combined-fair (proposed)
Aware application characteristics?	NO	YES	YES	YES
Implement on real machine?	YES	YES	NO	YES
Require extra hardware support?	NO	NO	YES	NO
Supported base performance	slow only	fair share	fast only	fair share slow only fast only
Guaranteed fairness metric	-	-	uniformity	minFairness uniformity

`scaled load balancing` proposed by Li et al. [7], [8] provides an equal computing capacity to each thread without considering how to support performance efficiency. It argues that if threads have the same priority, they should receive the same share of core processing power. Since cores differ in their processing capacity, the proposed scheduler adjusts CPU shares for different core types with a fixed scaling ratio. The ratio for load scaling is empirically obtained by the benchmark suite performance. However, this study focuses only on how to support fair shares of CPUs with a real system implementation, without considering throughput efficiency of asymmetric multi-cores.

Second, Kwon et al. discussed an equal share fairness definition [11], which is same as *max-fair* with *fair share* in our study. With their definition, the applications get the same chance to improve their performance by receiving the same share of fast cores. In addition, they also proposed an `R%-fair` scheduler, which runs applications fairly in $R\%$ of time, and uses `max-perf` policy for the rest of time quantum. Although it cannot guarantee a specific performance target, it attempts to improve fairness while still increasing the overall throughput. Using a scheduler implementation added to an open source hypervisor, the study showed the feasibility of such schedulers in virtualized systems.

Third, a study by Van Craeynest et al. investigated three fairness aware schedulers for AMP [13]. In this simulation-based study, they rely on an estimated performance of each application with an isolated fast core as the baseline performance to aim for the fairness. To obtain the estimated performance, they used a hardware-based performance model requiring some changes in performance monitoring [12]. They define fairness with the uniformity metric defined in the previous section. As a fair scheduling policy, they proposed an `equal-progress` scheduler, which provides an actual equal instruction throughput progress for each application by assigning appropriate shares of fast and slow cores. Their `guaranteed-fairness` scheduler aims to improve both throughput and fairness, by running as a throughput-maximizing scheduler until fairness drops below a given threshold, and as an `equal-progress` scheduler for the rest of scheduling period. However, this study does not discuss minimum fairness guarantee, and was conducted with architectural simulation with a special hardware change.

Last, this paper extends our prior work [18] in terms of supported base performance and guaranteed fairness metrics. We study three types of base performance for asymmetric multi-cores, and propose scheduling mechanisms to support complete fairness for each base performance. In addition, our scheduler can guarantee the specified level of uniformity as well as minimum fairness. Moreover, we further analyze `max-perf` policy, most prior work considered, by investigating the correlation of performance/fairness metrics with the application characteristics.

Table 2 compares the three prior schedulers to our schedulers proposed later in this paper. `Scaled load balancing` provides fair assignment of fast and slow core shares to support fairness, but it does not exploit different efficiencies of applications to improve the overall throughput. `R%-fair` adds the limited application-awareness to `scaled load balancing` by combining throughput and fairness goals with the selected R ratio. However, it does not provide any guaranteed fairness and how to select the R factor was not fully investigated. The `guaranteed fairness` scheduler warrants the uniformity as fairness, but requires new hardware supports for efficiency estimation. Their study is based on architectural simulation. The scheduler proposed in this paper can guarantee the target minimum fairness while improving both throughput and uniformity, being implemented for a real Linux system.

3.2 Other Schedulers

Most of the other prior work to investigate scheduling mechanisms for asymmetric multi-cores aim at maximizing system throughput [1], [6], [9], [10], [11], [12], with policies similar to `max-perf` in this paper. To pick the highest fast core efficiency applications, they use an exploration technique [1], architecture-independent signatures [6], and indirect estimation techniques using performance counters [9], [10], [11]. Craeynest et al. proposed a hardware-based approach to get an accurate fast core efficiency, but, it requires a special hardware and is highly dependent on the microarchitecture [12].

Another aspect of scheduling threads on asymmetric multi-cores is to support multi-threaded applications, and several prior studies attempted to improve the parallel scalability by running a bottleneck thread on a fast core. The identified bottlenecks are sequential phases [9], [19], delayed threads [20], and critical sections [21], [22].

For commercial processors targeting mobile systems such as the ARM big.LITTLE architecture, CPU utilization-based schedulers have been developed, as the mobile workloads exhibit severe fluctuations of CPU utilization.

4 Analysis of Throughput-Maximizing Policy

In traditional SMP, fairness in CPU provisioning can be achieved by adjusting the amount of CPU cycles of applications. At the same time, CPU utilization can be maximized by preventing cores from being idle while there are tasks to run. Throughput maximization and fairness support are independent problems and can be achieved simultaneously.

On the other hand, on AMP, not only amount of CPU cycles but also type of CPU cycles affect the application performance. Thus, throughput maximization and fairness support may not be achieved simultaneously.

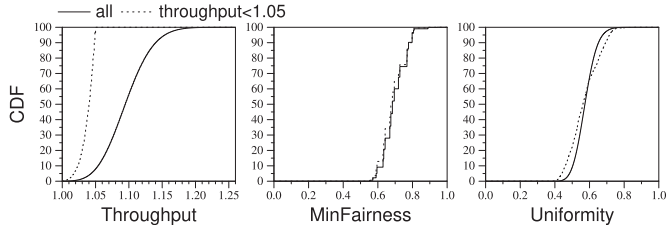

Fig. 1. Throughput and fairness of `max-perf`.

TABLE 3
Sample Correlation Coefficient (R-Value) Between the Fast Core Efficiency Distribution and the Results of `max-perf`

efficiency distribution	throughput	minFairness	uniformity
avg	−0.215	−0.855	−0.994*
stdev	0.858	−0.224	−0.231
stdev/avg	0.945*	0.114	0.136
third top	−0.204	−0.993*	−0.840

In this section, we analyze the throughput-maximizing scheduling policy, `max-perf`, proposed by most of the prior studies on asymmetric multi-cores. Both of throughput and fairness aspects are quantitatively analyzed with some ideal assumptions, and the causes of the results are discussed.

4.1 Methodology

The analysis of `max-perf` scheduling in this section is based on the following assumptions. First, we assume that the fast core efficiency is known for each application. In real systems, online fast core efficiency estimation may incur overheads and possible inaccuracy. Second, we assume that sufficiently fine-grained adjustment of CPU share is possible without any overhead. This implies two things: CPU usage ratio for fast and slow cores can be adjusted in any fine-grained way, and there is no overhead from context switch and thread migration. Third, there is no shared resource effect. While these assumptions are applied in the analysis in this section, our real machine implementation and evaluation in Sections 6 and 7 will remove the assumptions.

We model a hexa-core AMP, two fast cores and four slow cores, using the GEM5 simulator [23]. The fast core is a 4-way out-of-order processor, and the slow core is a single issue out-of-order processor. Each core has a 64 KB L1 instruction cache, 64 KB L1 data cache, and 2 MB L2 unified cache privately. Thus, there is no interference from cache sharing. To construct application mixes, we use all combinations with repetition of 23 SPECCPU 2006 benchmark. Since there are 6 cores, we use 6 applications for each mix and the number of mixes is ${}_{23}H_6 = 376,740$. For each application, the simulation skips 1 billion instructions with fast-forwarding, and runs 100 million instructions. For the base performance, *fair share* is used.

4.2 Fairness of Throughput-Maximizing Scheduling

Fig. 1 shows cumulative distributed function (CDF) with the throughput, minimum fairness, and uniformity results of `max-perf`. There are two lines per graph. Solid lines represent the results of all mixes. Dotted lines represent the results of mixes whose throughput improvement from `max-perf` policy is less than 5 percent, that is, whose throughput is less than 1.05.

For all mixes, represented as solid lines, `max-perf` shows performance improvement from the base performance up to 1.26 and the median is 1.09, as shown in the throughput graph. However, the strict `max-perf` scheduling often sacrifices minimum fairness and uniformity significantly. In the case of more than half, minimum fairness is lower than 0.7 and uniformity is lower than 0.6. The lower limit of minimum fairness is 0.54, and one of uniformity is 0.37. For the minimum fairness results, there are only 23 discrete minimum fairness levels, since 23 benchmark applications are used for this analysis, and thus there are the same number of normalized throughput levels without any interference by co-running. This perfect scheduling analysis does not have any random effect observed in real systems.

Note that the dotted lines are not much different from the solid lines, even though it represents only the cases that the throughput improvement from `max-perf` policy is less than 5 percent. This means that `max-perf` seriously lowers the fairness regardless of small throughput gains.

However, note that the base performance assumes that all applications have some amount of fast core share. `Max-perf` still improve system-wide throughput improvement up to 26 percent and the median of improvements is 9 percent. If we use `max-fair` policy, we lose this amount of throughput to support maximum fairness among applications.

Based on the observations in this section, this paper will relax the `max-perf` policy which is based on a strict efficiency order imposed even if the throughput gain is small. Our schedulers guarantee fairness metrics with a lower bound, and still improves the system-wide throughput.

4.3 Correlation between Fast Core Efficiency and Fairness

To analyze further, we investigate the correlation between the distribution of fast core efficiencies of applications in each mix and its throughput, minimum fairness, and uniformity results of `max-perf`. We use the sample correlation coefficients (R-value) for analysis, For the efficiency distributions, average, standard deviation, standard deviation divided by average, and third top are used. The R-value has values between -1 and 1. The absolute value of R means how two are strongly correlated. Also, R-value is positive if two are positively correlated, and is negative if two are negatively correlated.

Table 3 shows R-values for each fast core efficiency distribution and three metrics. The values with an asterisk mark indicate the highest absolute value for each column. Note that the highest absolute value for each column (metric) is more than 0.94. The fast core efficiency distributions we use explain the most portion of each metric.

As shown in the table, the throughput of `max-perf` is highly correlated with the standard deviation of efficiencies divided by the average of efficiencies. This means that `max-perf` gains much throughput when the applications have diverse characteristics but the average efficiency is low. The diversity of applications promises throughput improvement since it can exploit the diversity of core capabilities. Note that the standard deviation itself is also highly

correlated with the throughput. (Refer to the second row.) However, when the average efficiency is high, some high efficiency applications is scheduled on slow cores. Then, the applications show low performance compared to the base performance, which is throughput, and they lower the system-wide throughput, the average value of all application's throughput.

The third column of the table shows that minimum fairness is determined by the third top efficiency application. Since we use two fast cores and four slow cores, the third top efficiency application has the highest fast core efficiency among the applications receiving only slow core share by `max-perf`, and it shows the lowest throughput.

The uniformity is strongly correlated with the average efficiency. As seen before, the high average efficiency means that some high efficiency applications show low throughput. In addition, the applications running on fast core also have high efficiency, and show high throughput. Thus, the difference between high and low throughput becomes large, and so the uniformity does.

5 Design Space

In this section, we propose four fairness-oriented scheduling policies pursuing throughput improvement under fairness constraints. First, `min-fair` scheduling supports minimum fairness guarantee by restricting the maximum performance degradation from `max-fair`. The second scheduler, `uni-fair` supports uniformity as a guaranteed metric. It restricts the performance variance as the system administrator sets. The third scheduler, `sim-fair`, optimizes above two schedulers by equalizing the core share of applications which have similar fast core efficiency. Finally, `combined-fair` scheduling combines three scheduling policies above.

In this section, we use the notation in Table 1. In addition, we present the results with the same perfect scheduling assumptions used in Section 4. `Fair share` is used as the base performance for the analysis.

5.1 Minimum Fair Scheduling

The first scheduler, `min-fair` supports that a fixed level of throughput is always maintained. The administrator can set the maximum performance degradation (minF) compared to the base performance. For the given minF setting, the `min-fair` policy tries to improve throughput while supporting the strict minimum fairness of each application. To meet the minimum fairness requirement, every application is guaranteed to have a sufficient fast core share. After the minimum fairness is met, applications with the highest fast core efficiencies monopolize the remaining fast core shares. By doing this, `min-fair` guarantees $T^i \geq target$ for all i.

Algorithm 1 sketches the procedure of `min-fair` policy, which determines fast and slow core shares for each thread (f^i and s^i). Note that we use the notation in Table 1. The core part is **required_f_share()**. It calculates the amount of fast core shares which is required to guarantee its own minimum fairness target. To satisfy the minimum fairness limit, the inequality in the function must be held. The main algorithm begins from the scheduling of `max-fair`, to check whether the target is achievable or not. If the system administrator sets target which is not achievable, the scheduling uses `max-fair` policy. Then, it gives the portion of fast core share to all threads as much as calculated by **required_f_share()** so the minimum fairness limit is maintained. Then, the remaining portion is given to an application with the highest fast core efficiency to improve throughput. The maximum fast core share of an application is limited by the number of threads (1 thread per application in this work), and the process is repeated until the all taken fast core shares are distributed.

Algorithm 1. Min-Fair Policy

required_f_share(i, *target*)
  /* estimate the base performance */
  $perf_{base} \leftarrow e^i \times f^i_{base} + s^i_{base}$
  /* find the minimum fast core share to meet minF target */
  Find *f_minF* satisfying the following
    $\frac{e^i \times f\_minF + (c^i - f\_minF)}{perf_{base}} > target$
  return *f_minF*
sched_min_fair(*target***)**
  sched_max_fair()
  if calculate_minFairness() $<$ *target* **then**
    /* *target* is not achievable even with `max-fair` */
    return
  end if
  f_remain $\leftarrow F$
  for each i in all threads **do**
    share $\leftarrow$**required_f_share**$(i,$ *target*$)$
    $f^i \leftarrow$ *share*
    $s^i \leftarrow c^i -$ *share*
    f_remain $\leftarrow$ *f_remain* $-$ *share*
  end for
  for each i in descending order of fast core efficiency **do**
    share $\leftarrow$ **MIN**$(c^i - f^i, f\_remain)$
    $f^i \leftarrow f^i +$ *share*
    $s^i \leftarrow s^i -$ *share*
    f_remain $\leftarrow$ *f_remain* $-$ *share*
    if *f_remain* ≤ 0 **then**
      break
    end if
  end for

Fig. 2a shows the throughput results of `min-fair`(80 percent), `min-fair`(90 percent) and `max-perf`, and Fig. 2d shows the minimum fairness results of the same three configurations. As shown in the figures, `min-fair`(80 percent) and (90 percent) can effectively support the minimum fairness limit. Furthermore, with `min-fair`(80 percent), even if the system guarantees 80 percent performance from the `max-fair` state, it can gain throughput similar to `max-perf`. The `max-perf` policy may degrade minimum fairness by up to 60 percent, even with little throughput improvement (refer to Fig. 1). Such a modest target setting of 80 percent can prevent significant minimum fairness violations by `max-perf`. This result reinforces our initial observation that `max-perf` frequently sacrifices fairness severely, even if throughput gain is none or minor. Minimum fairness setting also results in uniformity improvement, as Fig. 2g.

Optimality. `min-fair` policy provides the optimal throughput under the minimum fairness constraints. We

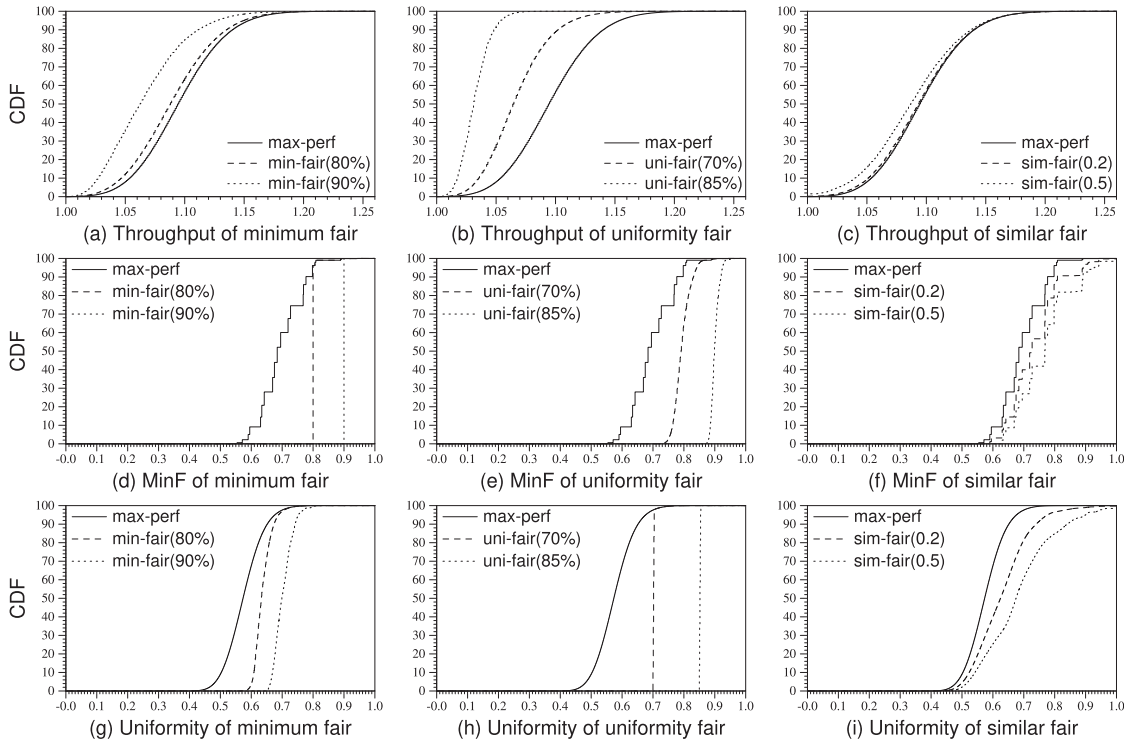

Fig. 2. CDF of throughput, minimum fairness, and uniformity with all possible 6 core combinations of 23 applications.

prove this by contradiction. Assume that there is an *optimal* core share distribution other than `min-fair`, which yields the higher throughput than `min-fair` and guarantee the minimum fairness over the target. In the *optimal* distribution, at least one application receives less fast core share than the distribution of `min-fair`. There are two cases for the application. The first case is that the fast core share of the application with `min-fair` is same with **required_f_share()**. Then, with the *optimal* distribution, the application receives fast core share less than **required_f_share()**. This makes the performance of the application less than the minimum performance requirements and breaks the minimum fairness target. The second case is that the fast core share of the application with `min-fair` is larger than **requilred_f_share()**. To yield the higher throughput, the fast core share from the application should be given to the higher fast core efficiency application. However, in the second loop of Algorithm 1, the remaining fast core share is given to applications in the descending order of fast core efficiency. Thus, there is no higher fast core efficiency application which can receive more fast core share. Therefore, there is no such *optimal* core share distribution.

5.2 Uniformity Fair Scheduling

The second scheduler, `uni-fair` restricts *uniformity*, which represents the inverse of performance variance, over the user-defined level. The administrator can set the maximum performance variance allowed as a uniformity value. For the given uniformity target, the `uni-fair` policy tries to improve throughput while guaranteeing the uniformity level of a system. To meet the uniformity requirement, the scheduler determines the portion of time `max-perf` can be used without hurting the uniformity requirement. Then, it runs as `max-perf` for the specified portion of time, and runs as `max-fair` for the rest of time. The concept of this scheduling is very similar to what Craeynest et al. [13] proposed. It calculates the uniformity of a system for each very short scheduling interval and determines what policy is used for the next interval. However, changing the scheduling at a fine-grained interval may be burdened on real machines, and, estimating the fast core efficiency at fine-grained interval is very hard without extra hardware support. Thus, our scheduler needs to assume long scheduling interval (2s), and calculates the exact portion of time that `max-perf` can be used to improve the throughput.

Algorithm 2 sketches the procedure of `uni-fair` policy. After calculating the uniformity with `max-perf` policy, it first assumes that the uniformity drops down linearly as the portion of time `max-perf` is used. With the assumption of the linear relationship, α, the portion of time `max-perf` is used, should be $(1 - target)/(1 - uniformity_{\texttt{max-perf}})$. The algorithm then finds the exact value of α by decreasing the value by 1 percent at a time until the uniformity estimated with α is greater than the target. At the end of the algorithm, fast core share of a thread is the weighted average of fast core share with `max-perf` and fast core share with `max-fair`, where the weight is α.

Fig. 2b shows the throughput results of `uni-fair`(70 percent), `uni-fair`(85 percent) and `max-perf`, and Fig. 2h shows the uniformity results of the same three configurations. As shown in the figures, `uni-fair`(70 percent) and (85 percent) can effectively support the uniformity limit. However, as strictly limiting the performance variance, `uni-fair` scheduling does not exploit the diversity of applications and the throughput gain drops down. Reducing the performance variance results in the improvement of minimum performance, and minimum fairness is also improved, as Fig. 2e.

Algorithm 2. Uni-Fair Policy

/* $f^i_{\texttt{max-perf}}$: fast core share of thread i by `max-perf` */
/* $f^i_{\texttt{max-fair}}$: fast core share of thread i by `max-fair` */
sched_uni_fair(*target*, $f_{\texttt{max-perf}}$, $f_{\texttt{max-fair}}$**)**
  /* calculate uniformity of `max-perf` */
  $f^i \leftarrow f^i_{\texttt{max-perf}}$ for all threads
  $s^i \leftarrow c^i - f^i_{\texttt{max-perf}}$ for all threads
  uniformity← **calculate_uniformity()**

  /* α is the time ratio of running with `max-perf` */
  /* Assume that uniformity is linearly related to α */
  $\alpha \leftarrow (1-$ *target*$)/(1-$ *uniformity*$)$
  for each i in all threads **do**
    $f^i \leftarrow f^i_{\texttt{max-perf}} \times \alpha + f^i_{\texttt{max-fair}} \times (1-\alpha)$
    $s^i \leftarrow c^i - f^i$
  end for
  uniformity ← **calculate_uniformity()**

  /* to find the exact value of α */
  while *uniformity* $<$ *target* **do**
    $\alpha \leftarrow \alpha - 0.01$ /* reduced by 1% */
    for each *i* in all threads **do**
      $f^i \leftarrow f^i_{\texttt{max-perf}} \times \alpha + f^i_{\texttt{max-fair}} \times (1-\alpha)$
      $s^i \leftarrow c^i - f^i$
    end for
    uniformity ← **calculate_uniformity()**
  end while

5.3 Similar Fair Scheduling

The third scheduler, `sim-fair` does not guarantee any fairness metric. Rather, it relaxes `max-perf` by equally distributing fast core shares to a group of applications with similar fast core efficiencies. Assuming there are N fast cores, in `max-perf`, the top N applications with the N highest efficiencies monopolize the fast cores. On the other hand, in `sim-fair`, the scheduler finds groups of applications whose fast core efficiencies are similar, with less than a *similarity* difference. The administrators can adjust the relaxation level by setting *similarity*. Then, it assigns an equal share of fast cores to every application in each group. However, across groups, their fast core shares may differ depending on the average fast core efficiencies of the groups. The fairness support in `sim-fair` attempts to reduce the negative artifact of the strict scheduling of the `max-perf` policy to improve uniformity, although it may potentially reduce the overall throughput.

Algorithm 3 presents the procedure of `sim-fair`. It starts from the core allocation used by the `max-perf` policy. At each scheduling interval, for threads receiving more fast core shares than the share assigned by the `max-fair` policy, threads with similar efficiencies are grouped together. Whenever the group formation is updated, the fast and slow core shares are updated for each application to the new average of fast and slow core shares in the group.

The rightmost column of Fig. 2 shows the results of `sim-fair`. Fig. 2c presents the throughput results. The throughput with `sim-fair` is slightly lower than that with `max-perf` for some cases, but the differences are relatively small. As the similarity setting gets smaller, the performance differences are reduced. As shown in Fig. 2f, `sim-fair` frequently improves minimum fairness as it distributes fast core share equally for all high fast core efficiency applications. Note that the minimum fairness highly depends on the *third top* fast core efficiency. Also, since `sim-fair` makes similar efficiency applications gets same chance to improve throughput, the uniformity is improved as shown in Fig. 2i.

Algorithm 3. Sim-Fair Policy

/* $f^i_{\texttt{max-perf}}$: fast core share of thread i by `max-perf` */
/* $f^i_{\texttt{max-fair}}$: fast core share of thread i by `max-fair` */
sched_sim_fair(*similarity*, $f_{\texttt{max-perf}}$, $f_{\texttt{max-fair}}$**)**
  /* start from `max-perf` schedule */
  $f^i \leftarrow f^i_{\texttt{max-perf}}$ for all threads
  $s^i \leftarrow c^i - f^i_{\texttt{max-perf}}$ for all threads

  for each i in all threads such as $f^i \geq f^i_{\texttt{max-fair}}$ **do**
    group = threads with efficiency difference $\leq$ *similarity*
    for each i in *group* **do**
      f^i = average f^i of threads in *group*
      s^i = average s^i of threads in *group*
    end for
  end for

5.4 Combined Fair Scheduling: Putting All Together

Finally, we propose `combined-fair` scheduler, which combines `min-fair`, `uni-fair`, and `sim-fair` schedulers. The administrators can set three parameters, similarity, minimum fairness, and uniformity. Then, `combined-fair` guarantees minimum fairness and uniformity greater than the targets and optimizes fairness based on the similarity parameter.

Algorithm 4 sketches how `combined-fair` combines three schedulers. First, it calculates the core share with `max-fair`, which is needed for `min-fair`. Second, `min-fair` policy works to guarantee the desired minimum performance level. Third, `combined-fair` calls `sim-fair` to consider the similarity parameter. However, `sim-fair` here starts from `min-fair` instead of `max-perf`, and the minimum fairness guarantee is kept. Last, `combined-fair` considers the uniformity target by doing what `uni-fair` does. One difference is that, for α portion of the time, the scheduler runs `min-fair + sim-fair`, instead of `max-perf`. Note that `max-fair` is used for the rest of time. This does not hurt the minimum fairness guarantee.

5.5 Discussion: Efficiency Estimation Error

One of the considerations for the algorithms is the tolerance of fast core efficiency estimation error, as the exact efficiency estimation is so difficult with specialized hardware support [12]. Fig. 3 shows the difference in fairness metrics with `min-fair` and `uni-fair` policy between the results with and without the efficiency estimation error. The results with the exact efficiency estimation are from the same method in the Fig. 2. Then, we add 5∼20 percent of the efficiency estimation error to the algorithm and get the other results. The results show that the minimum fairness

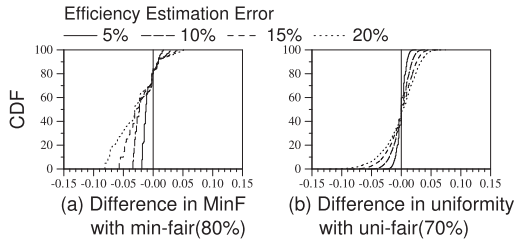

Fig. 3. Effect of efficiency estimation error on algorithms.

guarantee of `min-fair` is broken upto 3.4 percent with 10 percent efficiency estimation error, and the uniformity guarantee of `uni-fair` is broken upto 5.6 percent with 10 percent error in efficiency estimation. Note that our efficiency estimation mechanism on real machines show 8.82 percent error in the worst case (Section 7.5).

Algorithm 4. Combined-Fair Policy

sched_combined_fair(*similarity*, *minF_target*, *uniformity_target***)**

sched_max_fair()
$f^i_{\texttt{max-fair}} \leftarrow f^i$ for all threads

sched_min_fair(*minF_target* **)**
/* let $f_{\texttt{max-perf}}$ be the results of **sched_min_fair()** */
$f^i_{\texttt{max-perf}} \leftarrow f^i$ for all threads

sched_sim_fair(*similarity*, $f_{\texttt{max-perf}}$, $f_{\texttt{max-fair}}$**)**
/* let $f_{\texttt{max-perf}}$ be the results of **sched_sim_fair()** */
$f^i_{\texttt{max-perf}} \leftarrow f^i$ for all threads
sched_uni_fair(*uniformity_target*, $f_{\texttt{max-perf}}$, $f_{\texttt{max-fair}}$**)**

6 Implementation

We modified the Completely Fair Scheduler (CFS) scheduler of the Linux 3.10.96 kernel. The implementation requires three components. First, the schedulers must be able to estimate the fast core efficiency for each application online with as little overhead as possible. Second, the scheduler should control the usage of fast and slow core shares of applications to support fairness-oriented scheduling policies. Third, to avoid any performance overhead by scheduling two types of cores, it must be work-conserving, supporting that no core becomes idle when there are any pending ready threads. Furthermore, no fast core must become idle, when some tasks are running on slow cores, except for a very short transition period.

6.1 Online Fast Core Efficiency Estimation

One of the critical issues for scheduling threads on asymmetric multi-cores is to estimate the fast core efficiency of each thread. There have been several previous studies to estimate fast core efficiency [6], [9], [10], [11]. A common approach is to approximate the fast core efficiency based on instruction throughput or last-level cache (LLC) misses while an application is running on either a fast or slow core. The prior approaches assume that measuring fast core efficiencies by trying each thread on both types of cores is costly, as it requires to change core types periodically for each application. Such an approximation-based method may be able to provide approximate relative orders of efficiencies among applications. For the `max-perf` scheduling the prior estimation method is designed for, such rough ordering is good enough to determine which applications run on fast cores. However, to support the fine-grained fairness control as proposed in this paper, a more accurate estimation of fast core efficiency is necessary.

To support accurate estimation of fast core efficiencies, we use a direct method of measuring fast core efficiencies with an exploration-based approach. Instead of estimating fast core efficiencies with indirect metrics such as LLC misses, the proposed method measures the actual performance in both fast and slow cores by running threads on both cores periodically. A similar method with HW-based scheduling was proposed by Kumar et al. [1], and evaluated with simulation. We have implemented it on a Linux system, validating its cost is sufficiently low for real SW-based schedulers. Our fast core efficiency metric is as follows:

$$efficiency = \frac{IPS_{fast}}{IPS_{slow}}.$$

Instruction per second (IPS) is the primary metric of the performance, measured with common performance monitoring counters in commercial processors. For each scheduling interval, 2 seconds in our experiments, IPS on fast and slow cores are individually measured and averaged. Another benefit of this direct method is that it will work independently from the architectural characteristics of fast and slow cores. It measures the actual performance with fast and slow cores, instead of using an approximation.

There are three potential sources of overheads for the fast core efficiency estimation. First, to measure the actual performance on both core types, all threads should be scheduled on both types of cores for each scheduling interval. This forced scheduling can make applications run on less optimal core types occasionally. However, since the forced scheduling period for the estimation is short, the overhead is negligible. Second, this method adds more context switches even if an application should be scheduled to only one type of core continuously. Third, using performance monitoring units may have overheads. For optimization, we virtualized the performance monitoring unit, and directly read the machine specific registers (MSR) instead of counting the number of interrupts [24]. As will be discussed in Section 7.5, the proposed method can provide a highly accurate estimation with negligible overheads on a real machine.

6.2 Periodic Core Share Adjustment

Based on the estimated fast core efficiencies, our scheduler determines fast and slow core shares determined by three fairness policies. This occurs periodically, in our implementation, on every 2 seconds, and each share is written in the thread context. This process is implemented as a user-level program and it communicates with the kernel by syscalls. If this is implemented in the kernel, the overhead can be further reduced. However, Section 7.5 will show the share calculation overhead is negligible even with the user level implementation.

To support adjustable fast and slow core shares, we add *fast_round* and *slow_round* for each thread, which represent

TABLE 4
Workloads

Emulated AMP system		Big.LITTLE system	
Name	Benchmarks	Name	Benchmarks
HHH	povray×2, namd×2, bzip2×2	HH	gamess×2, bwaves×2
MMM	zeusmp×2, gcc×2, leslie3d×2	MM	h264ref×2, gromacs×2
LLL	soplex×2, mcf×2, milc×2	LL	gobmk×2, omnetpp×2
SAME	gcc×6	ML.a	bzip2×2, astar×2
MLL	gcc×2, omnetpp×2, mcf×2	ML.b	gromacs×2, sjeng×2
MML	gcc×2, leslie3d×2, milc×2	HM.a	GemsFDTD×2, h264ref×2
HMM	povray×2, gcc×2, leslie3d×2	HM.b	hmmer×2, gromacs×2
HHM	namd×2, hmmer×2, gcc×2	HL.a	GemsFDTD×2, omnetpp×2
HML.a	namd×2, gcc×2, soplex×2	HL.b	bwaves×2, gobmk×2
HML.b	h264ref×2, astar×2, omnetpp×2		
HHL.a	namd×2, hmmer×2, soplex×2		
HHL.b	gamess×2, gromacs×2, milc×2		
HLL.a	hmmer×2, mcf×2, milc×2		
HLL.b	gobmk×2, GemsFDTD×2, mcf×2		

how many rounds the thread has run on each type of cores. In addition, each thread has *fast_core_share* and *slow_core_share*. The fast or slow round is incremented whenever a thread completes to run on a fast or slow core for $fast\ or\ slow\_core\_share \times 30\,ms$ time period, respectively.

The scheduler forces each thread to use fast and slow cores as specified by fast and slow core shares, by maintaining that fast and slow core rounds proceed together. When a thread gets a timer tick on a fast core, the scheduler compares its *fast_round* with *slow_round*. If *fast_round* is greater than *slow_round*, the scheduler searches another thread on a slow core whose *fast_round* is less than *slow_round*. If such a thread is found, two threads swap their next core types to run.

7 Evaluation

7.1 Methodology

To evaluate the proposed fairness-oriented schedulers, we use two systems. The first one is an emulated AMP system. This system has a 6-core AMD Phenom II X6 1055T Processor. Asymmetric multi-cores are emulated by the DVFS mechanism. Two cores are configured to fast cores with their frequency set to 2.8 GHz, and the remaining 4 cores are set to slow cores with the frequency of 0.8 GHz. All six cores share a 6 MB last level cache. Although the emulated asymmetric cores differ only in their frequencies, this configuration exercises effectively both the scheduler and online fast core efficiency monitoring components in this study.

The second one is a real AMP system with the ARM big.LITTLE architecture. Our test platform is Odroid-XU3 Lite, which has the Exynos5422 SoC with four Cortex-A15 (big) cores and four Cortex-A7 (little) cores on a chip. Big cores are 3-way out-of-order cores running at 1.8 GHz, and little cores are 2-way in-order cores running at 1.3 GHz. Four big cores share a 2 MB L2 cache, and four little cores share another 512KB L2 cache. The device has two limitations. First, it does not fully support hardware performance monitoring units. Thus, we cannot use our online fast core efficiency estimation mechanism, and offline values are used. Second, it has only 2 GB DRAM, which is not sufficient to run 8 benchmarks on all the 8 cores. Thus, we use only two big cores and two little cores, and turn off the remaining cores.

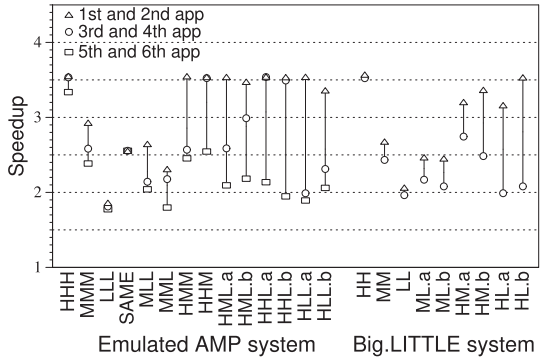

Fig. 4. Fast core efficiency distributions of workloads.

The workloads consist of mixes from SPECCPU2006, as shown in Table 4. The mixes for an emulated AMP system consist of 6 benchmarks as the system has 6 cores. We use the `reference` input sets on this system. On the other hand, the mixes for the big.LITTLE system have 4 applications and `train` input sets are used due to the limited DRAM capacity.

Fig. 4 presents the fast core efficiencies of applications in each mix we used for two systems. The efficiencies are measured by pinning an application on a fast or slow core for each mix. In the naming, *H*, *M*, and *L* stand for high, medium, and low efficiencies respectively. There are two same benchmark application for each letter. For example, *MLL* has two M-type applications which are *gcc* for both, and four L-type applications where two of them are *omnetpp* and two of them are *mcf*. One exception is the *SAME* mix, which includes six instances of *gcc*.

Even for the same benchmark application across mixes, the fast core efficiencies are different due to shared resource effects. For example, the efficiency of *milc* in *LLL* is 1.78, while the same application in *HHL.b* is 1.95. As the co-running applications can affect the actual fast core efficiency of an application, the online efficiency estimation is necessary as implemented in our schedulers.

We repeatedly run applications in a mix until all applications are finished at least once, to reduce the variability of experimental results. We use the execution time of the first run for the performance of each application. For the evaluation, we use the throughput and fairness metrics explained in Section 2.4.

7.2 Max-Fair and Max-Perf Behaviors

Before the proposed fairness-oriented schedulers are evaluated, this section presents the behaviors of the two baseline schedulers, `max-fair` and `max-perf`, comparing them against the Linux default scheduler (`unaware`), which is not aware of the uneven core capability. In addition, we also show a static scheduler (`static`), which binds each application to a core. For the static scheduling, we run three different mapping settings between applications and core types. The experimental results shown in this section use *fair share* base performance.

Fig. 5 presents the `unaware`, `static`, and `max-fair` results of *HML.b* and *HLL.b* workloads on the emulated AMP system. The remaining mixes show similar trends. In the figure, each bar represents the average throughput of a workload mix, and circles represent the throughputs of individual applications in the mix. The figure shows the

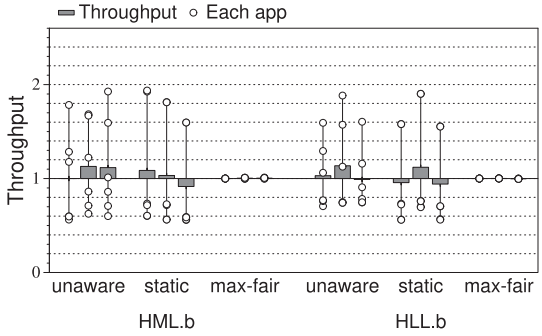

Fig. 5. Comparison of the max-fair scheduler to the default Linux scheduler.

results from three independent runs for each scheduler, and for `static`, each run uses a different affinity mapping. As shown in the figure, the `unaware` scheduler shows high variances in application throughputs, as the scheduler assumes symmetric multi-cores, resulting in random scheduling effects. The `static` scheduler also exhibits high variances in application performance for each different affinity setting, depending on what applications are pinned to fast cores. As the throughput of each application is normalized to that with the `max-fair` scheduler, the `max-fair` scheduler shows the throughput of 1 for all applications, without any significant random scheduling effect even in real runs.

Fig. 6 presents throughput results with `max-perf`. Each column corresponds to a different mix. For each column, empty circles on the line represent the throughputs of applications in the mix. The lowest throughput in the mix is the minimum fairness of the mix, and represented as a triangle. The bars show the system-wide throughput with each mix, and uniformity is also shown as a filled circle.

Although `max-perf` aims to maximize the throughput, it can achieve high throughput improvement when there are high fast core efficiency differences among applications. When applications in a mix have almost the same fast core efficiencies, such as *HHH*, *MMM*, and *LLL*, `max-perf` does not show noticeable throughput improvements. For workload mixes of diverse applications, `max-perf` improves 3∼13 percent of system-wide throughput. However, they frequently suffer from low uniformity and high minimum fairness degradation.

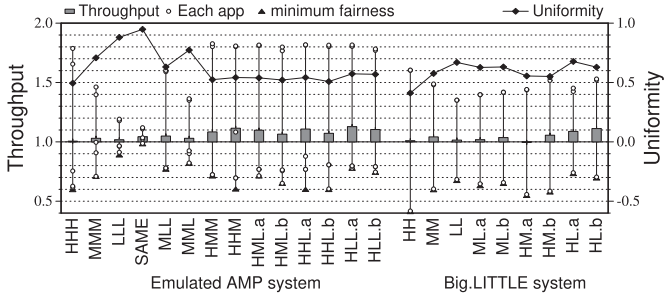

Fig. 6. Results of `max-perf`.

There are two exceptional cases. First, SAME mix consists of six copies of same worklod, *gcc*, but `max-perf` improves 4.3 percent of system-wide throughput. The main reason is that *gcc* consists of many phases with different fast core efficiency. Second, *HM.a* mix on the big.LITTLE system with `max-perf` shows slightly lower throughput than the result with `max-fair`. This is due to the clustered cache design of our experimental platforms. *HM.a* mix consists of two copies of a cache sensitive workload, *GemeFDTD*, and two copies of a cache insensitive workload, *h264ref*. With `max-fair` policy, all workloads run both types of cores and utilize two L2 cache clusters. However, `max-perf` binds the cache-sensitive workloads on one L2 cache cluster, limiting the throughput improvements.

7.3 Detailed Results

In this section, we present the effectiveness of the proposed three schedulers in the emulated AMP system. Figs. 7, 8 and 9 show the results for `min-fair`, `uni-fair` and `sim-fair` policies with *fair share* base performance. For each mix, each column corresponds to a different policy. The figures show normalized throughputs of all applications and the average of them, along with minimum fairness and uniformity results.

Fig. 7 shows the results of `min-fair` policy with the target minimum fairness level of 85, 90, and 95 percent. For comparison, it also shows the results of `max-perf`. First, the results show our implementation guarantees the specified minimum fairness level very effectively. Even for the case that `max-perf` degrades minimum fairness up to 60 percent, `min-fair(85 percent)` maintains minimum

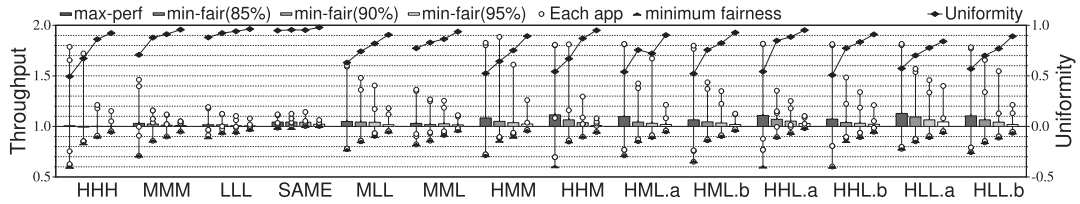

Fig. 7. Results of `minimum fair` on emulated AMP.

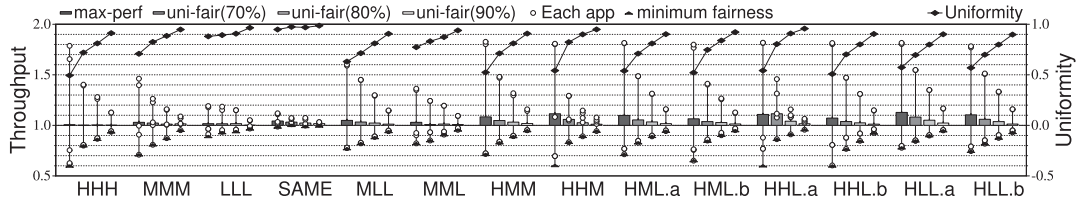

Fig. 8. Results of `uniformity-fair` on emulated AMP.

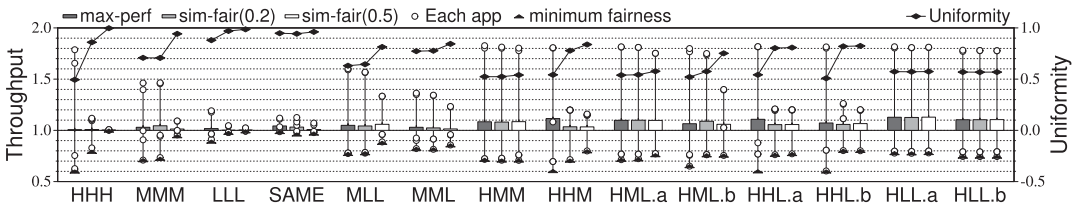

Fig. 9. Results of `sim-fair` on emulated AMP.

fairness higher than 85 percent. Similarly, `min-fair(90 percent)` and `min-fair(95 percent)` also effectively limit the performance degradation with the specified lower bound.

However, for the system-wide throughput, the figure shows the trade-offs in the throughput and minimum fairness. To support the higher level of minimum fairness, the system may exhibit the lower throughput for some mixes. The first six mixes, *HHH* to *MML*, show little throughput degradation, mostly smaller than 2 percent, with large minimum fairness and uniformity improvements. Since difference on fast core efficiencies of applications in the mixes, giving the required fast core share to all applications does not affect the overall throughput. However, for the rest mixes, throughput is degraded up to 6 percent to meet the target minimum fairness level. The main reason is the coexistence of *H* and *L* applications. The low fast core efficiency applications need some amount of fast core shares to guarantee the minimum fairness level, but the fast core shares given to such applications do not contribute effectively to the throughput improvement. Moreover, stealing fast core shares from *H* largely hurts the throughput significantly.

One positive effect of supporting minimum fairness is the improvement of uniformity. Throughout the mixes, uniformity is improved significantly compared to `max-perf`. However, uniformity is not a guaranteed metric. For example, the uniformity of *HML.a* with `min-fair(90 percent)` is lower than `min-fair(85 percent)` in spite of its higher minimum fairness target. For *HML.a*, more than 100 percent fast core share is required to support higher than 85 percent minimum fairness target. Then, `min-fair` policy picks up only one application, with the highest fast core efficiency, to give all remaining fast core share. In common cases, there is a little disturbance in fast core efficiency estimation, and the highest rank in fast core efficiency occasionally changes between two copies of H application. Unfortunately, when the experiment with `min-fair(90 percent)` runs, only one copy of them monopolizes the remaining fast core share until ends. `Min-fair` policy does not consider this and shows low uniformity.

Fig. 8 shows the results of `uni-fair` policy with the target uniformity level of 70, 80, and 90 percent. For comparison, it also shows the results of `max-perf`. First, the results show our implementation guarantees the specified uniformity level very effectively, even for the case that `max-perf` degrades minimum fairness up to 52 percent. In addition, if `max-perf` already shows the high uniformity, such as *LLL* and *SAME*, `uni-fair` keeps the uniformity level of `max-perf`.

As seen in *MML*, *HHM*, and *HHL.a*, `uni-fair` conservatively guarantee the target uniformity level. since our implementation guarantees the uniformity target for each 2 seconds interval. For each interval, the scheduler estimates the uniformity from `max-perf`. If it seems to be higher than the target, the scheduler runs as `max-perf` and gets the higher uniformity. Otherwise, `uni-fair` makes the uniformity level same as the target. Therefore, the overall uniformity can be higher than the target. This situation does not occur on the big.LITTLE system, as we use the offline fast core efficiency and do not change the core share of applications during the entire run.

Fig. 9 shows the results of `sim-fair` with *similarity* of 0.2 and 0.5. The results of `max-perf` policy are also shown for comparison. First, workload mixes which include similar fast core efficiency applications, such as *HHH*, *HHM*, *HHL.a*, and *HHL.b*, benefit from `sim-fair` with 0.2 similarity. Except for *HHM*, the policy mostly improves the uniformity with little change in performance. In addition, *sim-fair* with 0.5 similarity works for *HHH*, *MMM*, *MLL*, and *HML.b*, as more workloads in the mixes can be grouped with 0.5 similarity. On the other hand, *sim-fair* shows neither throughput nor uniformity changes for the rest, since it fails to make any groups due to the large difference in fast core efficiencies among applications.

For *MMM* and *HML.b*, `sim-fair` with 0.2 similarity shows higher throughput than `max-perf` policy. This is due to the small error from our fast core efficiency estimation mechanism. Since our implementation uses the estimated fast core efficiencies, `max-perf` policy may run the second highest efficiency applications on fast cores. However, as `sim-fair` re-distributes the fast core share among high fast core efficiency applications, the applications with actually the highest efficiency get the chances of running on fast cores. Note that the improvement is small and the error is fixed with small (0.2) similarity. Our fast core efficiency estimation mechanism show high accuracy as shown in Section 7.5. Finally, `sim-fair` with 0.5 similarity again degrades the system-wide throughput since it is likely to give more fast core share to actually low efficiency applications.

In summary, `min-fair` can provide a fixed performance lower bound, although setting the lower bound very high can degrade throughput significantly for some mixes. Even with a relatively modest minumum fairness restriction of 85 percent, `min-fair` can avoid critical performance degradations from `max-perf` which are up to 60 percent. `Uni-fair` can limit a performance variance under a specified level, but it shows the trade-off between the uniformity level and the throughput improvement. Last, `sim-fair` improves uniformity effectively without any significant effect on the overall throughput, except for two cases.

7.4 Summarized Results

In this section, we summarize all results from both of emulated AMP and real AMP systems with three base performances, *fast only*, *slow only*, and *fair share* base. We use target

TABLE 5
Target Parameters for Evaluation (the Values with Asterisks Are Used for `combined-fair` Scheduling.)

	fair share	fast only	slow only
similarity		0.2* 0.5	
minFairness	85%* 90%, 95%	40%* 50%, 60%	120%, 140%* 160%
uniformity		70%, 80%, 90%	

parameters listed in Table 5 for `sim-fair`, `min-fair`, and `uni-fair` scheduling. For `combined-fair` scheduling, we use the parameters with asterisk marks in the table for similarity and minimum fairness, and all of listed uniformity parameters. Thus, the number of parameter combinations for `combined-fair` scheduling is 3 for each base performance. Finally, since the number of mixes is 14 for emulated AMP system and 9 for real AMP system, the number of experiments is 462 for emulated AMP system and 297 for real AMP system.

For evaluation of the fairness aspects, the most important thing is whether the desired target metrics is achieved or not. We represent the amount of target achievement by the following metric

$$achievement_{fairness} = \frac{value}{\mathbf{MIN}(target, value_{\texttt{max-fair}})}.$$

In the equation, $value$ is a minimum fairness or uniformity value, $value_{\texttt{max-fair}}$ is a value of the same metric with `max-fair` policy. Note that when *fast only* or *slow only* base performance is used, the maximum value of minimum fairness and uniformity depends on the fast core efficiency distribution. If the specified target is larger than the maximum value, our achievement metric uses the maximum achievable value instead of the target.

Fig. 10 shows CDF of fairness target achievements for all experiments. The experiments with `sim-fair` policy is omitted as the policy does not guarantee any metrics. For `combined-fair` scheduler, both of minimum fairness target achievement and uniformity target achievement are included for each mix. The graph shows that our schedulers effectively guarantee the target metrics. On the emulated AMP, which is represented as a solid line, the worst case achievement is 95 percent. In addition, 97 percent of the experiments achieve more than 98 percent of target metrics. On the big.LITTLE system, the worst case achievement is 88 percent, and only 87 percent of the experiments achieve more than 98 percent of target metrics.

Such low achievements mainly come from the experiments with *fast only* and *slow only* base. This is related to the clustered cache design in our experimental platform. When measuring base performance for throughput calculation, we pin each application in a mix to a core and measure the execution time. This makes each application utilize only one of the cache clusters. However, with proposed scheduling, applications are likely to have core share from both types of cores. This causes cache interference, which is not included in the base performance. Note that *fair share* base performance already includes the cache interference effect. With *fair share* base, the minimum achievement increases to 93 and 97 percent of the experiments achieves more then 98 percent of target.

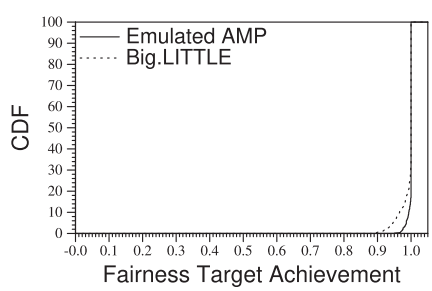

Fig. 10. CDF of fairness target achievements.

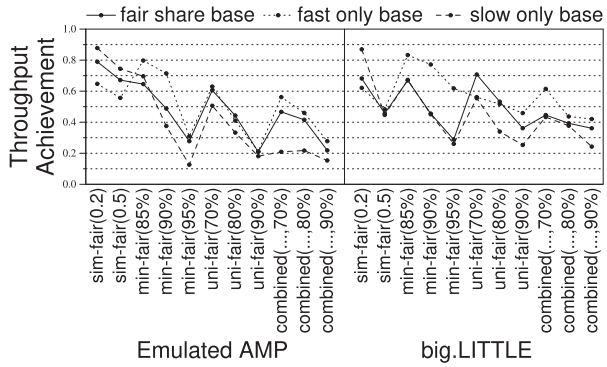

Fig. 11. Throughput achievements (average of all mixes).

To evaluate the throughput aspects, we define an $achievement_{throughput}$ metric as follows:

$$achievement_{throughput} = \frac{T - T_{\texttt{max-fair}}}{T_{\texttt{max-perf}} - T_{\texttt{max-fair}}}.$$

The throughput is minimum with `max-fair` policy and maximum with `max-perf` policy. Thus, this metric represents how much throughput is gained, as a percentage of the maximum throughput we can get.

Fig. 11 shows the results. Each column represents a parameter for a scheduling policy, and each line represents the base performance as labeled. Circles in lines show the average of throughput achievements from all mixes used for each system. The graph shows the trade-off between throughput and fairness in asymmetric multi-core systems. To guarantee higher fairness level, the more fast core share should be used only for fairness guarantee. Thus, the less fast core share can be used to improve throughput. In our benchmarks, guaranteeing 85 percent minimum fairness with fair share base means losing about 35 percent of throughput gain, and guaranteeing 70 percent uniformity with fair share base means losing 30-40 percent of throughput gain.

7.5 Accuracy and Scheduling Overhead

To evaluate the accuracy of fast core efficiency estimation, we compare the estimated fast core efficiency from `max-fair` and `max-perf` scheduler with the real fast core efficiency, which is measured by pinning an application on a fast or slow core for each mix. Since we use fair share base performance, all applications receive enough fast core share and slow core share with `max-fair`. Thus, `max-fair` can provide the most stable results for efficiency estimation. On the other hand, `max-perf` always gives the minimum core share for one of the types, and it may cause the most

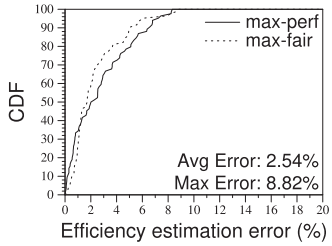

Fig. 12. Difference between estimated efficiency and real efficiency from pinned runs.

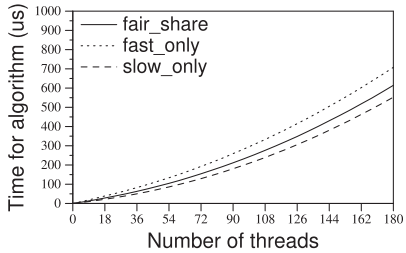

Fig. 14. Time for `combined-fair` algorithm.

unstable results. Also, for estimated efficiency, we average the values from all intervals until one application finishes once.

Fig. 12 shows the results. It presents the cumulative distribution function of estimation error in percentage. Two lines indicate `max-fair` and `max-perf` policy as labeled. The average error between the estimated efficiencies and measured ones is only 2.54 percent and the maximum error 8.82 percent. Our exploration-based efficiency estimation accuracy is high enough for supporting minimum fairness.

In addition, scheduling interval length may affect the accuracy of fast core efficiency estimation, since the information from the previous interval is used for the estimation of efficiency of the next interval. Fig. 13 shows the efficiency variation between the consecutive intervals for different scheduling interval lengths. We measure the efficiency for each interval by running the workload mixes in our experiments with `max-fair` policy. As in the figure, the short interval length shows the fluctuated efficiencies. However, if the interval is too long, our scheduler may react too slowly to the program phase change. Thus, we choose 2 seconds as the scheduling interval.

To assess scheduling overheads, we first compare the native Linux and `max-fair` schedulers with the fast core efficiency estimation on symmetric cores with the same 2.8 GHz clock speed. This comparison represents the pure overhead of frequent context switches and efficiency estimation procedures. Compared to the native Linux, the maximum throughput difference is 2 percent in the worst case.

Second, we measure the CPU time our scheduler uses. This includes the CPU time used for our scheduler itself, such as the time for estimating fast core efficiencies, processing algorithms for our policies, and handling syscalls. In the worst case, the CPU usage time is less than 0.53 ms. Since there are 6 cores and the scheduling interval is 2 seconds, the overhead on CPUs is less than 0.0044 percent.

Despite such low overhead, some of our algorithms are not scalable on the number of threads. To estimate the overhead with a large number of threads, we measure the CPU time for the algorithms with different numbers of threads. The measurement is done on slow cores, running at 0.8 GHz, for the worst case overhead. We omit the time for efficiency estimation and syscall handling since they can be done in parallel for each thread. Fig. 14 shows the results with `combined-fair` algorithm with 90 percent target uniformity, which requires the largest CPU time. Although the time for algorithms increases as the number of threads increases, the required time is less than 710 us with 180 threads. If there are 180 cores for 180 threads, the worst case overhead is less than 0.0002 percent. Moreover, note that our algorithms are not related to the correctness of program running. It can be delayed for a while until any core becomes idle.

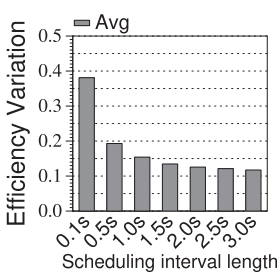

Fig. 13. Efficiency variation for different interval lengths.

7.6 Discussion: Extend to Multi-Threaded Applications

As mentioned in Section 2.1, the proposed schedulers do not consider the relationship between threads in multi-threaded applications. Instead, they manage each thread independently as most OSes do. However, if schedulers consider such relationship on asymmetric multi-core systems, the performance of applications can be improved without additional core share.

There are two issues for scheduling multi-threaded applications on asymmetric multi-core systems. The first issue is how to define fast core efficiency of the application. Since the fast core efficiencies of threads in an application may differ, it is hard to estimate the benefit of fast core share for multi-threaded applications. Second, intra-application scheduling is also the important issue. Unlike single thread applications, not only the fast core efficiency of a thread, but also the criticality of the thread affect the performance of multi-threaded applications. For example, the thread on a critical section may be the most beneficial thread for receiving fast core share, even if its fast core efficiency is lower than other threads. Some papers have investigated such issues to optimize the multi-threaded application performance on AMP [13], [20], [22], [25].

Our scheduler design can be easily extended to support multi-threaded applications. Our work aims at the fairness of inter-application scheduling, and the extension should address intra-application scheduling. To show the feasibility, we implement a prototype extension with the simplest methods. For the fast core efficiency of an application, our prototype uses the average of threads' fast core efficiencies. Also, for intra-application scheduling, we apply `max-fair` policy for threads in an application. Although this method ignores the criticality of threads, this gives an illusion of

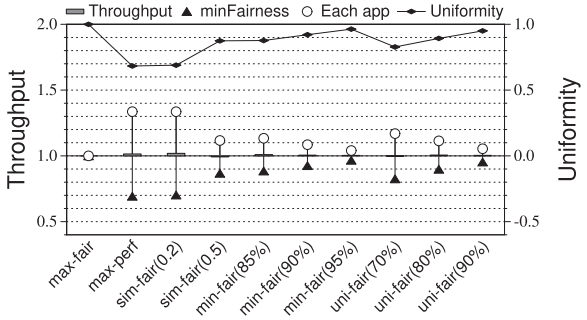

Fig. 15. Results of extension for multi-threaded applications.

symmetric multi-cores for threads, which is the assumption in programmers' mind.

Fig. 15 shows sample results with *swaptions* and *facesim* from PARSEC benchmarks. Each application uses three threads and their fast core efficiencies are 3.58 for *swaptions* and 3.09 for *facesim*. The x-axis presents the scheduling policies with different parameters, and the y-axis presents throughput and fairness results. The results show that our prototype extension guarantees the inter-application fairness with the multi-threaded applications. However, the implementation may not work if applications have heavy synchronizations. Our future work is to develop a rigorous extension to maximize the throughput of multi-threaded applications by intra-application scheduling, while our current work still guarantees the inter-application fairness.

8 Conclusions

This paper investigated fair scheduling support for asymmetric multi-core systems with two different aspects, minimum fairness and uniformity. The analysis concludes that the prior throughput-maximizing scheduler often sacrifices minimum fairness and uniformity excessively to gain only a small amount of throughput. To mitigate the problem, this paper proposed `min-fair`, `uni-fair`, and `sim-fair` schedulers, to guarantee minimum fairness or uniformity. Also, we proposed `combined-fair` which combines the benefits of three schedulers. We modified a Linux scheduler to support the fair scheduling policies and experimentally showed that the schedulers can support fairness with negligible performance overheads.

Acknowledgments

This work is supported by the National Research Foundation of Korea (NRF-2016R1A2B4013352) and by the Institute for Information & communications Technology Promotion (IITP-2017-0-00466). Both grants are funded by the Ministry of Science and ICT, Korea.

References

[1] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2003, pp. 81–92.

[2] N. K. Choudhary, et al., "FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template," in *Proc. Int. Symp. Comput. Archit.*, 2011, pp. 11–22.

[3] P. Greenhalgh, "Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7," *ARM Whitepaper*, 2011.

[4] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proc. Int. Symp. Comput. Archit.*, 2008, pp. 363–374.

[5] S. Herbert and D. Marculescu, "Variation-aware dynamic voltage/frequency scaling," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2009, pp. 301–312.

[6] D. Shelepov, et al., "HASS: A scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, pp. 66–75, 2009.

[7] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2007, pp. 1–11.

[8] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2010, pp. 1–12.

[9] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proc. Eur. Conf. Comput. Syst.*, 2010, pp. 139–152.

[10] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. Eur. Conf. Comput. Syst.*, 2010, pp. 125–138.

[11] Y. Kwon, C. Kim, S. Maeng, and J. Huh, "Virtualizing performance asymmetric multi-core systems," in *Proc. Int. Symp. Comput. Archit.*, 2011, pp. 45–56.

[12] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proc. Int. Symp. Comput. Archit.*, 2012, pp. 213–224.

[13] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 177–187.

[14] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proc. USENIX Symp. OS Des. Implementation*, 1996, pp. 107–122.

[15] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors," in *Proc. Symp. Operating Syst. Des. Implementation*, 2000, Art. no. 4.

[16] A. Chandra and P. Shenoy, "Hierarchical scheduling for symmetric multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 3, pp. 418–431, Mar. 2008.

[17] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proc. Int. Symp. Comput. Archit.*, 2004, pp. 64–75.

[18] C. Kim and J. Huh, "Fairness-oriented OS scheduling support for multicore systems," in *Proc. Int. Conf. Supercomput.*, 2016, Art. no. 29.

[19] J. C. Saez, A. Fedorova, M. Prieto, and H. Vegas, "Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors," in *Proc. ACM Int. Conf. Comput. Frontiers*, 2010, pp. 31–40.

[20] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–12.

[21] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multicore architectures," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2009, pp. 253–264.

[22] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 223–234.

[23] N. Binkert et al."The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[24] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *Proc. Int. Symp. Comput. Archit.*, 2011, pp. 353–364.

[25] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric CMPs," in *Proc. Int. Symp. Comput. Archit.*, 2013, pp. 154–165.

Changdae Kim received the BS, MS, and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST). He is a research fellow in computer science with the Korea Advanced Institute of Science and Technology. His research interests include computer architecture, operating systems, and cloud computing.

Jaehyuk Huh received the BS degree in computer science from Seoul National University, and the MS and PhD degrees in computer science from the University of Texas at Austin. He is an associate professor of computer science with the Korea Advanced Institute of Science and Technology (KAIST). His research interests include computer architecture, parallel computing, virtualization, and system security. He is a member of the IEEE.