Fairness-oriented OS Scheduling Support for Multicore Systems

Changdae Kim
School of Computing, KAIST
cdkim@calab.kaist.ac.kr

Jaehyuk Huh
School of Computing, KAIST
jhhuh@kaist.ac.kr

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 the multi-cores with uneven computing capability, 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 multi-cores with uneven capability, 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 three scheduling policies which guarantee a minimum performance bound while improving the overall throughput and reducing performance variation too. The three 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.

CCS Concepts

•Software and its engineering → Scheduling; •Computer systems organization → *Multicore architectures;*

Keywords

fair scheduling, asymmetric multicore, performance variance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICS '16, June 01-03, 2016, Istanbul, Turkey

© 2016 ACM. ISBN 978-1-4503-4361-9/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2925426.2926262

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 [9, 2]. Furthermore, process variation incurs different maximum frequencies for cores in a multi-core [20, 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 multi-cores with uneven capability, scheduler support is crucial. While scheduling for asymmetric multi-cores has been widely studied [9, 18, 14, 15, 17, 8, 11, 22, 21, 20], most of the studies aim at maximizing overall throughput by exploiting uneven capability and application behaviors. Such throughput-maximizing scheduling assigns fast cores to applications with high relative performance gains with fast cores compared to slow cores.

However, an important but neglected aspect of CPU scheduling in the prior studies is fairness of CPU provisioning. Unfair scheduling may not meet deadlines of real-time applications [13]. 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 [14, 15, 11, 21], 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 uneven 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 uneven cores. Prior throughput-maximizing

schedulers often sacrifice fairness of CPU provisioning excessively to gain only a small amount of extra throughput.

This paper proposes three new fairness-oriented schedulers for multi-cores with uneven computing capability, which allow a certain level of fairness to be guaranteed while improving throughput. The first scheduler, `sim-fair`, reduces performance variation of the prior throughput-maximizing scheduler by relaxing the strict throughput-oriented allocation. It attempts to reduce the variance, although no fixed performance lower bound is guaranteed. The second 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 third scheduler, `sim-min-fair`, combines the benefits of the previous two schedulers. It supports minimum performance guarantee while reducing performance variation. The three different fairness-oriented schedulers provide the system administrator with the mechanisms to choose different ways of setting fairness requirements. All three 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 3.7.3 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 (DVFS). 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. Although only the relative order of fast core speedup is sufficient for the prior throughput-oriented scheduling, providing fairness guarantee requires a more accurate fast core speedup estimation. 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 [4]. The results with various mixes show that our schedulers guarantee the specified fairness and still improve the overall throughput.

The main contributions of this paper compared to prior work are as follows.

- Unlike prior fairness studies for uneven cores focusing on reducing performance variance, the proposed scheduler supports minimum fairness guarantee, to strictly limit the performance degradation beyond the allowed level. This study investigates two different aspects of fairness, minimum performance and performance variation, while prior studies aim to solve only one of the two aspects.

- The proposed schedulers are implemented in the Linux system to prove that the schedulers can support fairness in a real machine. The implementation proves that measuring accurate fast core speedups directly by running applications on both types of cores periodically is feasible.

- The proposed purely software-based mechanism does not require any extra modification of existing architectures.

Limitation: The implemented scheduler is fully running on multi-cores with two different frequencies by DVFS. In the result section, we also validate our implementation on a real AMP with two types of cores. However, the real AMP evaluation does not support the online speedup estimation technique due to the lack of support for the hardware monitoring counters in the architecture.

The remainder of the paper is organized as follows. Section 2 discusses fairness in multi-cores with uneven computing capability and analyzes fairness of throughput-maximizing scheduling. In Section 3, we propose three fairness-oriented scheduling policies. Section 4 describes the implementation issues including the fast core speedup estimation mechanism. The experimental results on real machines are shown in section 5. Section 6 presents the related work, and Section 7 concludes this paper.

2. FAIRNESS FOR UNEVEN CORE CAPABILITY

In traditional homogeneous multi-cores, fairness in CPU provisioning can be achieved by adjusting shares of CPU cycles for each application. At the same time, CPU utilization can be maximized by preventing cores from being idle while there are tasks to run. However, on multi-cores with uneven computing capability, throughput maximization and fairness support may not be achieved simultaneously. Cores have different computing capabilities, and the performance improvement from multiple types of cores varies across different applications.

Most of the prior studies on uneven cores 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. This section defines the fairness aspects for scheduling uneven cores. To simplify analysis and discussion in this paper, we use two types of cores, *fast* and *slow cores* in a multi-core architecture.

2.1 Definitions and Metrics

In this section, we define throughput, and two fairness metrics, which will be used for the rest of paper. In addition, we also describe two scheduling policies, `max-perf` which maximizes only the throughput and `max-fair` which maximizes only the fairness as two opposite ends.

We define the *fast core speedup* of an application as the relative performance on a fast core compared to that on a slow core. If the performance of an application is defined as its execution time, the fast core speedup of an application is defined as follows. $exec\_time_{fast}$ or $exec\_time_{slow}$ is the execution time of an application, when the application is running entirely on a fast core or on a slow core.

$$speedup = \frac{perf_{fast}}{perf_{slow}} = \frac{exec\_time_{slow}}{exec\_time_{fast}}$$

To support fairness in real systems, it is necessary to set a performance baseline with a *fair* scheduling state. Since the fair state performance must be measurable in running systems, we use the following definition as the *fair* state as proposed by Kwon et al. [11]: *a scheduling in a system with fast and slow cores is fair, if all threads receive equal shares*

of fast and slow cores. Since this definition solely depends on the number of core cycles, neither prior knowledge on application behaviors nor any performance model is required. The proposed system will use the performance of each application in their fair state as the normalization baseline for throughput and fairness.

Alternatively, the fair state can be defined based on the slowdown compared to an isolated run without any co-runner. Threads are *fairly* scheduled if the slowdowns compared to the isolated runs of all threads are equal, as used by Van Craeynest et al. [21]. However, to achieve the fair state with this definition, the scheduler needs prior knowledge about the performance in the isolated run, and this cannot be measured dynamically in running systems without new hardware supports and estimation methods.

With our definition of the fair state, we define T_i, the *throughput* of an application i as the performance normalized to the *fair* state performance. For the *system-wide throughput* metric, T, we use the arithmetic mean as follows.

$$T_i = \frac{perf_i}{perf_{i,fair}} = \frac{exec\_time_{i,fair}}{exec\_time_i} \qquad T = \frac{1}{n}\sum T_i$$

With this metric, we define `max-perf` scheduling which maximizes the system-wide throughput without considering fairness. Suppose the number of fast cores is N. Then, `max-perf` selects the N applications with the highest fast core speedups, 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 fair state performance of each application. Uniformity is how uniform the performances of applications are relatively to the fair state performances respectively. Minimum fairness, the minimum performance relative to the *fair* state, is defined as follows. Note that T_i is the performance normalized to the fair state performance.

$minF = min(T_1, T_2, ..., T_n)$ where there are n applications.

Second, the uniformity metric is the fairness metric proposed by Van Craeynest et al. [21]. It is based on a standard deviation of normalized performance of each application. For the normalization point, they use an estimated isolated performance on fast cores. Uniformity is defined as follows.

$$Uniformity = 1 - (\sigma_{T_i}/\mu_{T_i})$$

In this equation, σ_{T_i} is the standard deviation of application throughputs, and μ_{T_i} is the average of application throughputs. Unlike the original study [21], in this paper, the throughput is the performance normalized to the *fair* state, not to the isolated state.

Two schedulers proposed in this paper guarantee the mandated minimum fairness level. However, uniformity is improved, but not guaranteed. We decided to choose minimum fairness as the guaranteed fairness over uniformity unlike the prior study [21], since being able to set the lower bound of performance degradation is more critical than reducing the overall performance variation.

Finally, `max-fair` scheduling provides the fair state scheduling. It achieves the fair state by giving an equal share of fast and slow core cycles to all active threads. In this case, the throughputs of every thread are 1 by the definition of throughput. Thus, the minimum fairness is 1 and the uniformity is also 1 with `max-fair`.

2.2 Prior Fairness-aware Schedulers

There have been three studies to incorporate fairness into the scheduling problem of uneven cores. First, `scaled load balancing` proposed by Li et al. [14] 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. To exploit the efficiency provided by uneven computing capability, schedulers must also be aware of different fast core speedups of applications and improve overall throughput while supporting fairness.

Second, Kwon et al. discussed an equal share fairness definition [11]. They defined the fair state as the threads in the system receive equal shares of both types of cores, as used for our study. Since each application receives the same share of fast cores, the applications get the same chance to improve their performance. In addition, they also proposed an `R%-fair` scheduler, which runs applications fairly in $R\%$ of time, and uses `max-perf` scheduling 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 in virtualized systems, the study showed the feasibility of such schedulers.

Finally, a recent study by Van Craeynest et al. investigated three fairness aware schedulers for AMP [21]. 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 [22]. However, since such extra hardware supports are not available in real systems currently, we use the equal-share fairness as our baseline performance. 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.

Table 1 compares the three prior schedulers to our schedulers proposed later in this paper. `Scaled load balancing` provides application independent fair assignment of fast and slow core shares to support fairness. However, it does not utilize different speedups of applications to improve the overall throughput while enhancing fairness. `R%-fair` adds the limited application-awareness to the `scaled load bal-`

	scaled load balancing [14]	R%-fair [11]	Guaranteed-fairness [21]	sim-min-fair (proposed)
Aware application characteristics?	NO	YES	YES	YES
Estimate speedup at runtime?	NO	YES	YES	YES
Implement on real machine?	YES	YES	NO	YES
Require extra hardware support?	NO	NO	YES	NO
Guaranteed fairness metric	-	-	uniformity	minimum fairness

Table 1: Comparison with prior approaches

ancing scheme by combining throughput and fairness goals with the selected R ratio for different applications, although it does not provides any guaranteed fairness. How to select the R factor for different applications was not fully investigated. The `guaranteed fairness` scheduler warrants the uniformity as fairness, but requires new hardware supports for performance 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.

2.3 Fairness of Max-perf Policy

In this section, we discuss how much minimum fairness and uniformity are supported in the perfect `max-perf` scheduling. For perfect scheduling, we first assume that the fast core speedup is known for each application. In real systems, online fast core speedup estimation may incur overheads and possible inaccuracy. Second, we assume that sufficiently fine-grained adjustment of CPU share is possible without any overhead. This assumption 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 or 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 Section 4 and 5 will remove the assumptions.

We model a hexa-core AMP, two fast cores and four slow cores, using the GEM5 simulator [1]. 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 64KB L1 instruction cache, 64KB L1 data cache, and 2MB L2 unified cache privately. Thus, there is no interference through cache sharing. To construct benchmark application mixes, we use all combinations with repetition of 23 SPECCPU 2006 benchmark. Since there are 6 cores, the number of total mixes is ${}_{23}H_6 = 376740$. For each application mix, the simulation skips 1 billion instructions with fast-forwarding, and runs 100 million instructions. Performance is normalized to that with `max-fair`.

Figure 1 shows the throughput, minimum fairness, and uniformity results of `max-perf`. Each workload mix is a combination of 6 applications running on the 6-core AMP. The workload mixes are presented in their throughput order. In the bottom two graphs for minimum fairness and uniformity, workload mixes are also sorted by the same throughput order as the top throughput figure to investigate the correlation between throughput and fairness. The minimum fairness results show 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.

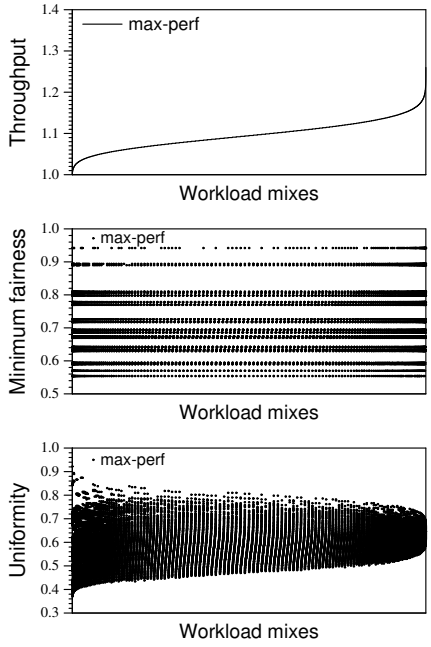

Figure 1: Throughput and fairness of `max-perf`

The strict `max-perf` scheduling often sacrifices minimum fairness and uniformity significantly to gain a small amount of throughput improvement, as shown in the left side of the graphs. Even when the throughput gain is less than 3%, minimum fairness can be degraded as much as 55% and uniformity can drop to less than 0.3. Throughput and minimum fairness do not exhibit any clear correlation. Regardless of throughput gains, minimum fairness is affected by the speedup characteristics of applications in the same mix.

Based on this observation, this paper will relax the `max-perf` policy which is based on a strict speedup order imposed even if speedup differences are small. Furthermore, a fair scheduler must be able to limit the performance degradation with a lower bound to prevent wide minimum fairness variations.

3. DESIGN SPACE

In this section, we propose three fairness-oriented scheduling policies pursuing throughput improvement under fairness constraints. First, `sim-fair` scheduling relaxes the throughput maximization goal to improve uniformity. The second scheduler, `min-fair` supports minimum fairness guarantee by restricting the maximum performance degradation from `max-fair`. The system administrator sets a minimum fairness level, and the scheduler makes scheduling decisions satisfying the constraint. The scheduler maximizes the throughput under the fairness constraint as much as possible. The third scheduler, `sim-min-fair`, combines the two approaches into a scheduler, pursuing both minimum fairness and uniformity.

Algorithm 1 sim-fair policy

sched_similar(*similarity***)**
  /* start from `max-perf` schedule */
  sched_max_perf()

  /* f_share: fast core share for a thread */
  /* s_share: slow core share for a thread */
  /* f_share_{fair}: fast core share by `max-fair` */

  for each *thr* in all threads such as $thr.f\_share \geq f\_share_{fair}$ **do**
    group = threads with speedup difference $\leq similarity$
    for each *thr* in *group* **do**
      $thr.f\_share$ = average f_share of threads in *group*
      $thr.s\_share$ = average s_share of threads in *group*
    end for
  end for

In this section, we present the results with the same perfect scheduling assumptions used in Section 2.3.

3.1 Similar Fair Scheduling

The first scheduler, `sim-fair` relaxes `max-perf` by distributing fast core shares equally to a group of applications with similar fast core speedups. Assuming there are N fast cores, in `max-perf`, the top N applications with the N highest speedups monopolize the fast cores. On the other hand, in `sim-fair`, the scheduler finds groups of applications whose fast core speedups 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 speedups 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 1 presents the procedure of `sim-fair`, which determines fast and slow core shares for each thread (f_share and s_share). 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 speedups 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 leftmost column of Figure 2 shows the results of `sim-fair`. Each line is independently sorted in the curves of all graphs in the figure. Figure 2a 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 Figure 2g, `sim-fair` frequently improves uniformity with minor throughput degradations. The performance variance highly depends on the high fast core speedup applications scheduled on slow cores by `max-perf`. To mitigate this, `sim-fair` gives such applications chances to receive some fast core shares. The disadvantage of `sim-fair` is that it cannot guarantee any minimum fairness constraint, as shown in figure 2d. The next `min-fair` scheduling will allow the system administrator to set the minimum fairness constraint.

Algorithm 2 min-fair policy

extra_f_share(*thr, target***)**
  /* return fast core share unnecessary to meet minF target */
  /* estimate performance under max-fair */
  $perf_{fair} \leftarrow core_{slow,fair} + thr.speedup * core_{fast,fair}$

  /* find the minimum fast core share to meet minF target */
  Find f_share_{minF} satisfying the following
    $\frac{(1 - f\_share_{minF}) + thr.speedup * f\_share_{minF}}{perf_{fair}} > target$
  /* return extra fast core share */
  return $thr.f\_share - f\_share_{minF}$

sched_min_fair(*target***)**
  /* start from `max-fair` schedule */
  sched_max_fair()

  $donated\_f\_share \leftarrow 0$
  for each *thr* in all threads **do**
    $amount \leftarrow$**extra_f_share**$(thr, target)$
    $thr.f\_share \leftarrow thr.f\_share - amount$
    $thr.s\_share \leftarrow thr.s\_share + amount$
  end for

  for each *thr* in descending order of fast core speedup **do**
    $amount \leftarrow$**MIN**$(1 - thr.f\_share, donated\_f\_share)$
    $thr.f\_share \leftarrow thr.f\_share + amount$
    $thr.s\_share \leftarrow thr.s\_share - amount$
    $donated\_f\_share \leftarrow donated\_f\_share - amount$
    if $donated\_f\_share \leq 0$ **then**
      break
    end if
  end for

3.2 Minimum Fair Scheduling

The second scheduler, `min-fair` supports that a fixed level of throughput is always maintained. The administrator can set the maximum performance degradation (minF) compared to that with the `max-fair` scheduling. 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 speedups monopolize the remaining fast core shares. By doing this, `min-fair` guarantees $T_i \geq target$ for all i.

The algorithm 2 sketches the procedure of `min-fair` policy. The core part is **extra_f_share()**. It calculates the amount of fast core shares that a donor can give to other applications without hurting 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`. Then, it takes the portion of fast core share from all threads as much as calculated by **extra_f_share()** so that the minimum fairness limit is maintained. Then, the taken portion is given to an application with the highest fast core speedup 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.

Figure 2b shows the throughput results of `min-fair`(80%), `min-fair`(90%) and `max-perf`, and Figure 2e shows the minimum fairness results of the same three configurations. As

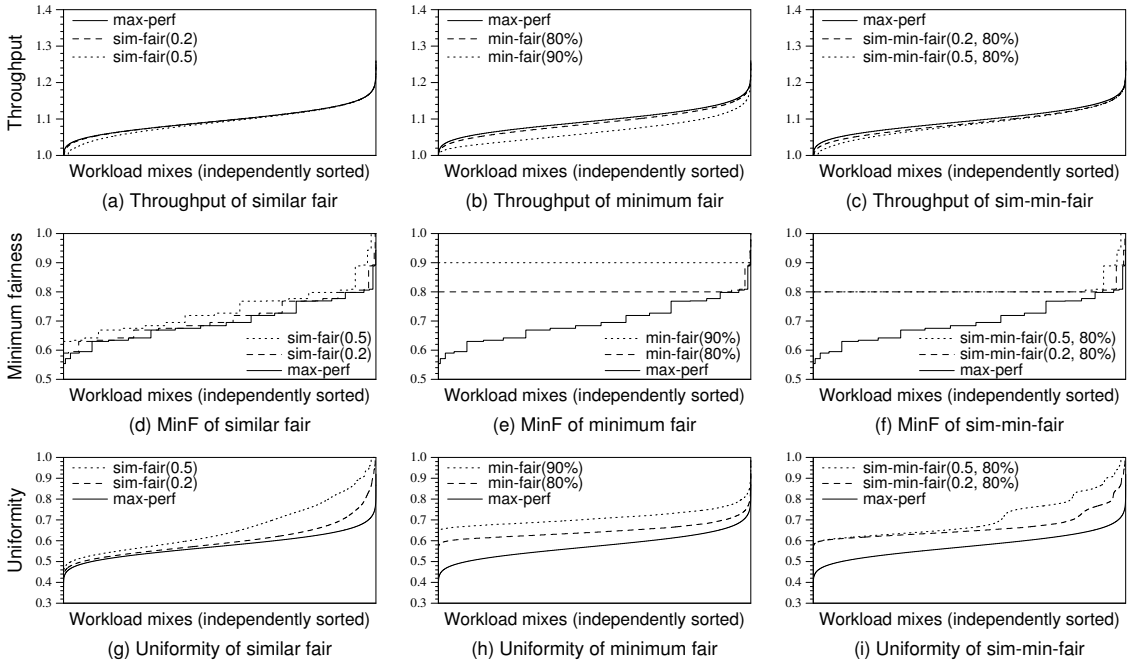

Figure 2: Throughput, minimum fairness, and uniformity distributions with all possible 6 core combinations of 23 applications

shown in the figures, `min-fair`(80%) and (90%) can effectively support the minimum fairness limit. Furthermore, with `min-fair`(80%), even if the system guarantees 80% 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%, even with little throughput improvement (refer to figure 1). Such a modest target setting of 80% 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 figure 2h.

3.3 Similar Minimum Fair Scheduling

The former two schedulers have different objectives. The `sim-fair` policy mitigates the large performance variance when there are many applications with similar characteristics, and the `min-fair` policy guarantees the minimum throughput for all applications. In this section, we combine the two policies into `sim-min-fair`.

`sim-min-fair` has only one modification from `sim-fair`. This starts from `min-fair` instead of `max-perf`. Similarly to `sim-fair`, the policy re-assigns the portion of fast core shares which exceed the completely fair shares. Since the fast core shares for supporting minimum fairness are always lower than or equal to the completely fair shares, the redistribution of fast core shares never degrades the fairness level already supported by `min-fair`. Thus, no application will hurt their minimum fairness, and applications with similar fast core speedups receive the same fast core share.

Figures 2c, 2f, and 2i show the analysis results. The minimum fairness target is set to 80% and *similarity* is varied with 0.2 and 0.5. Figure 2c shows that `sim-min-fair` has a minor performance degradation compared to `max-perf`. With little performance degradation, the minimum fairness is always guaranteed to the specified level as shown in Figure 2f. Sometimes minimum fairness is higher than the constraint since the integrated *similarity* setting distributes fast core shares evenly for high fast core speedup applications. Combining two fairness-oriented schedulers also shows the further improvement on uniformity as shown in Figure 2i.

4. IMPLEMENTATION

We modified the CFS (Completely Fair Scheduler) scheduler of the Linux 3.7.3 kernel. The implementation requires three components to support similar and minimum fairness constraints. First, the schedulers must be able to estimate the fast core speedup for each application online with as little overhead as possible. Second, the scheduler should periodically adjust fast and slow core shares of applications to guarantee fairness requirements and maximize throughput under the fairness constraints. Third, the scheduler requires a mechanism to assign different shares of fast and slow cores dynamically to each thread. 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.

4.1 Online Fast-core Speedup Estimation

One of the critical issues for scheduling threads on uneven cores is to estimate the fast core speedup of each thread. There have been several previous studies to estimate fast core speedups [18, 17, 8, 11]. A common approach is to approximate the fast core speedup 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 speedups by trying each thread on both types of core 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 speedups 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 speedup is necessary.

To support accurate estimation of fast core speedups, we use a direct method of measuring fast core speedups with an exploration-based approach. Instead of estimating fast core speedups with indirect metrics such as LLC misses, the proposed method measures the actual performance in terms of instruction throughput 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. [9], and evaluated with simulation. The study assumes that an architectural mechanism schedules and measures speedups with low overheads. We have implemented it on a Linux system, validating its cost is sufficiently low for real SW-based schedulers. Our fast core speedup metric is as follows.

$$speedup = \frac{IPS_{fast}}{IPS_{slow}}$$

IPS (instruction per second) 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 approximation.

However, there are three potential sources of overheads for the fast core speedup estimation. First, to estimate fast core speedups, all threads should be scheduled on both types of cores for each scheduling interval. This forced scheduling can potentially make applications run on less optimal core types occasionally. However, since the forced scheduling period for the fast core speedup 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 counters may have overheads, as it incurs interrupts frequently. To optimize this, we virtualized the performance monitoring unit, and directly read the machine specific registers (MSR) instead of counting the number of interrupts [3]. As will be discussed in Section 5.4, the proposed method can provide a highly accurate estimation with negligible overheads on a real machine.

4.2 Periodic Core Share Adjustment

Based on the estimated fast core speedups, our scheduler determines fast/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 5.4 will show the share calculation overhead is negligible even with the user level implementation.

To support adjustable fast and slow core shares, we have added $fast\_round$ and $slow\_round$ for each thread, which represent 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 $slow\_core\_share * 30ms$ 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 proceeds 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.

5. EVALUATION

5.1 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. Among 6 cores, 2 cores are configured to fast cores with their frequency set to 2.8GHz. The remaining 4 cores are set to slow cores with the frequency of 0.8GHz. Each core has a private 64KB instruction and data cache with a 512KB unified L2 cache. All six cores share a 6MB L3 cache. Although the emulated asymmetric cores differ only in their frequencies, this configuration exercises effectively both the scheduler and online fast core speedup 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 1.8GHz, and little cores are 2-way in-order cores running at 1.3GHz. Four big cores share a 2MB 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 speedup estimation mechanism. Instead, offline values are used with this system. Second, it has only 2GB 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. We ported our scheduler to Linux 3.10.96 to run it on this device.

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. The fast core speedups in Figure 3 are the average of estimated fast core speedups for each interval. 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.

Figure 3 presents the fast core speedups of applications in each mix we used for two systems. In the naming, *H*, *M*, and *L* stand for high, medium, and low speedups respectively. Approximately, *H* applications have a fast core speedup larger than 3, and *L* applications have one smaller than 2.4. There are two same benchmark application for each letter. For example, *MLL* has two M-type applications which are *gcc*

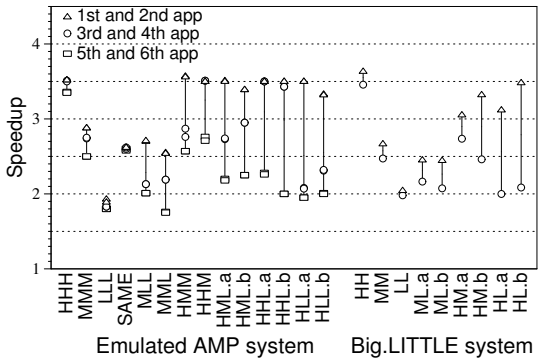

Figure 3: Speedup distributions of workloads

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		

Figure 4: Workloads

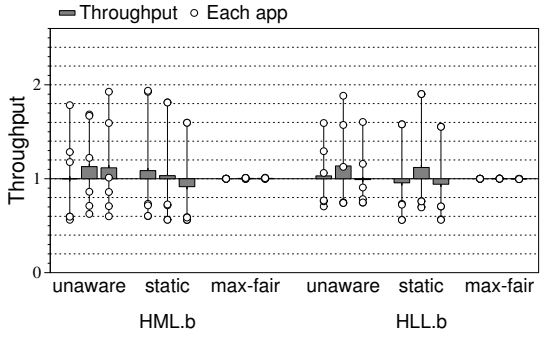

Figure 5: Comparison of the max-fair scheduler to the default Linux scheduler

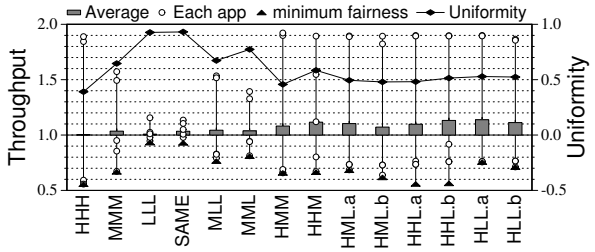

Figure 6: Results of `max-perf`

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 speedups are different due to shared resource effects. For example, the speedup of *milc* in *LLL* is 1.82, while the same application in *HHL.b* is 2.03. As the co-running applications can affect the actual fast core speedup of an application, the online speedup 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 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.

In the rest of this section, we first present the results from the emulated AMP system since the system can fully support the proposed mechanisms. Section 5.5 shows the results from the big.LITTLE system with the offline speedup values.

5.2 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 experiments use the emulated AMP system.

Figure 5 presents the `unaware`, `static`, and `max-fair` results of *HML.b* and *HLL.b* workloads. 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 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 the applications, without any significant random scheduling effect even in real runs.

Figure 6 presents normalized throughput results with `max-perf`. For each mix, each column corresponds to a different mix. For each column, empty circles on the line represent the normalized throughputs for all applications in the mix. The lowest throughput among the applications in the mix is minimum fairness of the mix. The bar in each column shows the average throughput of the mix. On top of the throughput results, uniformity is also shown as a filled circle in the figure. Note that the throughput metric is the performance normalized to the `max-fair` policy.

Although `max-perf` aims to maximize the throughput, it can achieve high throughput improvement when there are high speedup differences among applications. Due to

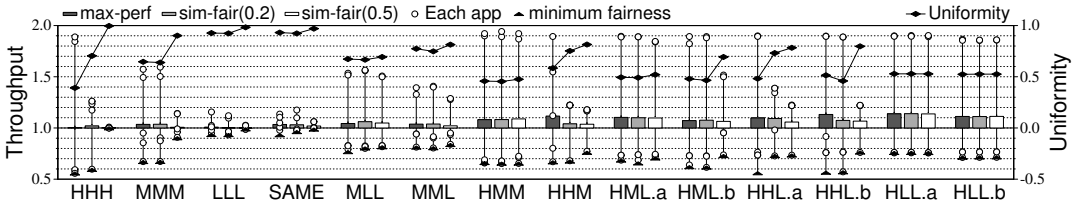

Figure 7: Results of `similar fair` on emulated AMP

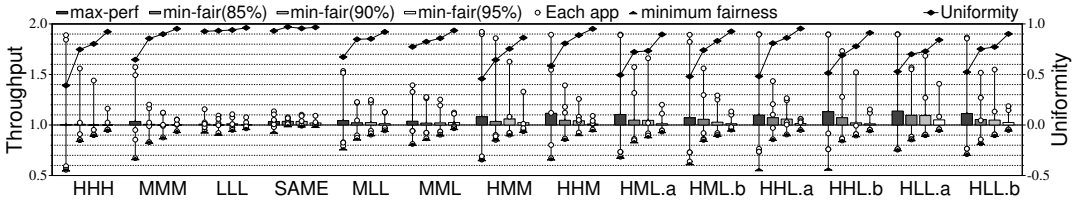

Figure 8: Results of `minimum fair` on emulated AMP

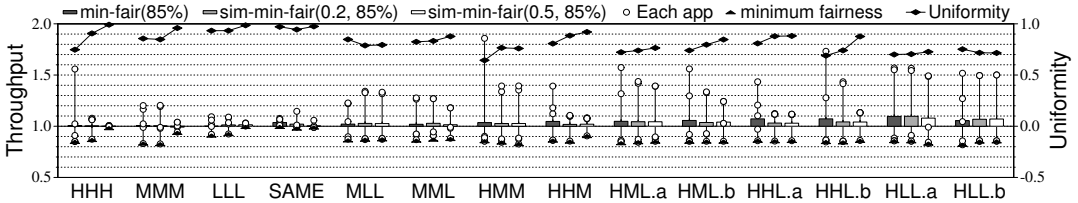

Figure 9: Results of `sim-min-fair` on emulated AMP

the behavior of `max-perf`, *HHH*, *MMM*, and *LLL* mixes do not exhibit noticeable throughput improvements from `max-fair`, since their applications have almost the same fast core speedups. For *HHH*, some applications exhibit 1.89X improvement, but they are offset by the performance degradation in other applications. For other workload mixes, `max-perf` improves 3∼14% system-wide throughput, and up to 1.92X for *povray* in *HMM*. Although `max-perf` can improve throughput when applications in a mix have high differences in speedups, they frequently suffer from low uniformity and high minimum fairness degradation.

5.3 Fairness-oriented Scheduling Results

In this section, we present the effectiveness of the proposed three schedulers in the emulated AMP system. Figures 7, 8 and 9 show the results for `sim-fair`, `min-fair` and `sim-min-fair` policies. 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.

Figure 7 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 with similar fast core speedups, such as *HHH*, *MMM*, *LLL* and *SAME*, benefit from `sim-fair` with 0.5 similarity. In these cases, the policy mostly improves the uniformity with little change in performance, `sim-fair` treats applications with similar speedups as a group to avoid unnecessary sacrifice of uniformity for little throughput gain. Similarly, in *HHM*, and *HHL.b*, 4 high applications have very similar fast core speedups, and thus `sim-fair` can improve uniformity effectively with little throughput degradation. In the mixes, the average throughput is not significantly affected since the performance loss in the top applications is offset by the performance gain in the second top applications. Finally, *MLL*, *HMM*, *HML.a*, *HLL.a*, and *HLL.b* show neither throughput nor uniformity changes with `sim-fair`. Since the fast core speedup differences in the mixes are large, `sim-fair` does not form any similar groups, and cannot improve uniformity. In summary, `sim-fair` does not significantly degrade the overall throughput except for *HML.b* and *HHL.a*, but uniformity can be improved significantly for the mixes where `max-perf` is not effective for improving throughput.

Figure 8 shows the results of `min-fair` policy with the target minimum fairness level of 85%, 90%, and 95%. 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%, `min-fair(85%)` maintains minimum fairness higher than 85%. Similarly, `min-fair(90%)` and `min-fair(95%)` 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. We can divide the results of `min-fair(85%)` in two groups. The first six mixes, *HHH* to *HHM*, show little throughput degradation, mostly smaller than 3%, with large minimum fairness and uniformity improvements. However, the last six mixes, *HML.a* to *HLL.b*, show mostly 4%∼7% throughput degradations to meet the target minimum fairness level. The main reason is the co-existence of *H* and *L* applications. The low fast core speedup 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 improve-

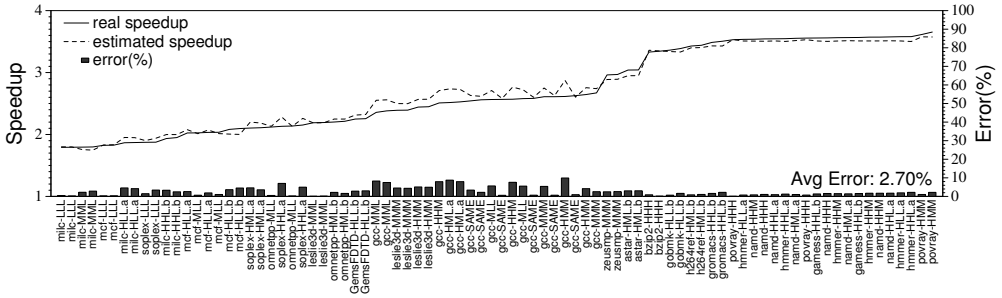

Figure 10: Comparison of estimated speedup and real speedup from pinned runs

ment. Moreover, stealing fast core shares from *H* largely hurts the throughput significantly. One positive effective of supporting minimum fairness is the improvement of uniformity. Throughout the mixes, uniformity is improved significantly compared to `max-perf`.

Figure 9 shows the results with varying *similarity* with `sim-min-fair`. The target minimum fairness level is fixed at 85%. Between `min-fair` and `sim-min-fair`, there are no significant throughput changes, except for *HHM*. However, adding the similarity factor improves the uniformity effectively without any significant performance degradation. `min-fair` is based on the fast core donation, and the donee application is picked from the ones with the highest fast core speedups. Even if two applications have similar speedups, only one of them (one with the slightly higher speedup) is picked as donee, receiving more fast core shares. It can degrade uniformity significantly for such mixes. However, `sim-min-fair` solves the problem by equalizing the fast core share among applications with similar fast core speedups.

In summary, `sim-fair` improves uniformity effectively without any significant effect on the overall throughput, except for two cases. `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 minimum fairness restriction of 85%, `min-fair` can avoid critical performance degradations from `max-perf` which are as large as 60% in some cases. Finally, `sim-min-fair` can have both of the benefit of similarity grouping to improve uniformity, and the benefit of minimum fairness support to limit performance degradation.

5.4 Accuracy and Scheduling Overhead

To evaluate the accuracy of speedup estimation, we compare the estimated speedup from `max-fair` scheduler and the real speedup, which is measured by pinning an application on a fast or slow core for each mix. Figure 10 shows the results. The x-axis represents applications in mixes, and the lines represent the estimated and real fast core speedups. The bars indicate the percentage of errors between the real and estimated speedups. Note that the speedup of an application changes depending on co-running applications due to the effect of cache and memory bandwidth sharing, and thus the same application appears multiple times in the x-axis with different mixes. The average error between the estimated speedups and measured ones is only 2.73% and the maximum error 10.04%. Our exploration-based speedup estimation accuracy is high enough for supporting minimum fairness.

To assess scheduling overheads, we first compare the native Linux and `max-fair` schedulers with the speedup estimation on symmetric cores with the same 2.8GHz clock speed. This comparison represents the pure overhead of frequent context switches and speedup estimation procedures. Compared to the native Linux, the maximum throughput difference is 2% 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 speedups, processing algorithms for our policies, and handling syscalls. In the worst case, the CPU usage time is less than 0.17ms. Since there are 6 cores and the scheduling interval is 2 seconds, the overhead on CPUs is less than 0.002%.

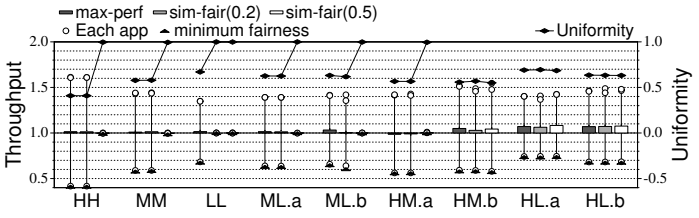

Figure 11: Results of `similar fair` on Big.LITTLE

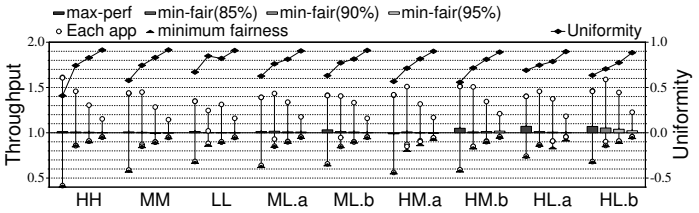

Figure 12: Results of `minimum fair` on Big.LITTLE

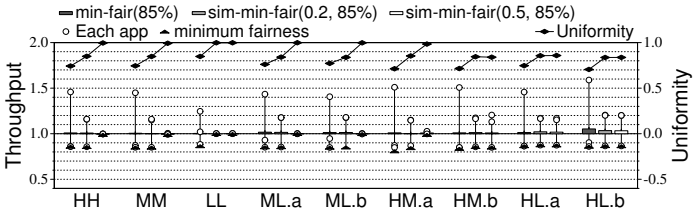

Figure 13: Results of `sim-min-fair` on Big.LITTLE

5.5 Big.LITTLE System Results

This section presents the effectiveness of fairness scheduling in a real AMP system. Figures 11, 12, and 13 present the results for three scheduling policies on the big.LITTLE system. The figures use the same notations as those in Section 5.3. The results on the big.LITTLE system also show the effectiveness of the proposed schedulers, as in the previous subsections. However, there are minor differences.

First, the throughput improvements of `max-perf` are up-to 7%, which is smaller than the improvement on the emulated AMP system. The main reason is the smaller differences in the speedups of applications in each mix. The mixes for the big.LITTLE system have two pairs of applications. Thus, the diversity of applications is lower than the mixes for the emulated AMP system with three pairs of applications.

Second, while our scheduler guarantees the target minimum fairness very effectively for most of the mixes, there is a minor exception. For example, a *GemsFDTD* instance in *HL.a* mix with the target minimum fairness of 90% shows the throughput of 83.5%. The missed fairness target is due to the clustered cache design in the big.LITTLE system and static speedup parameters used for this big.LITTLE configuration. Due to the same fixed speedups for two *GemsFDTD* instances used in this experiment, one instance of *GemsFDTD* gets a higher share of big core than the other instance of the same application by their IDs. Due to the contention in the big core shared cache caused by the first *GemsFDTD* instance, the performance of the second instance is slowed down significantly, violating the minimum fairness constraint. With `sim-min-fair`, such problems do not occur, as the two instances receive similar fast core shares due to the similarity of their fast core speedups.

Except for the aforementioned minor differences, this limited evaluation on the big.LITTLE system also validates that our proposed fairness-oriented scheduling works properly. One future work from this study is to investigate the effect of separate shared caches.

6. RELATED WORK

Apart from the fairness-aware scheduling studies discussed in Section 2.2, there have been studies on asymmetric multi-core processor architectures and their schedulers. Kumar et al. proposed an asymmetric multi-core processor (AMP) and showed its potential to improve area and energy efficiency [9, 10]. Recently, AMPs have been realized in academic and commercial designs. FabScalar project proposed RTL designs to compose asymmetric cores [2], and ARM released the big.LITTLE architecture [4].

In addition to the architectural exploration of AMP designs, there have been several studies to investigate scheduling mechanisms for AMPs. Most of the prior studies proposed schedulers to maximize system throughput [9, 18, 17, 8, 11, 22] with policies similar to `max-perf` in this paper. To pick the highest fast core speedup applications, they use an exploration technique [9], architecture-independent signatures [18], and indirect estimation techniques using performance counters [17, 8, 11]. However, to support fairness, it is necessary to estimate an accurate speedup for each application, examining whether the fairness is violated or not. Craeynest et al. proposed a hardware-based approach to get an accurate fast core speedup, but, it requires a special hardware and is highly dependent on the microarchitecture [22].

Another aspect of scheduling threads on uneven 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 [16, 17], delayed threads [12], and critical sections [19, 6]. Recently, Joao et al. proposed a utility-based acceleration mechanism, which considers all types of bottlenecks [7]. Utility is the amount of application level performance improvement when the thread is accelerated by a fast core. Although our schedulers do not directly evaluate multi-threaded applications, using utility as a fast core speedup metric for multi-threaded applications is our future work.

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. `HMP scheduler` uses the runnable time ratio of each thread to distinguish whether a thread has a sufficient potential to exhibit high CPU utilization. It runs threads with high runnable time ratios (high CPU utilization) on big cores, while migrating threads with low runnable time ratios to slow cores.

7. CONCLUSIONS

This paper investigated fairness of CPU provisioning in uneven cores 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 `sim-fair`, `min-fair`, and `sim-min-fair` schedulers, to improve uniformity and/or guarantee minimum fairness. We modified a Linux scheduler to support the three fair scheduling policies, and experimentally showed that the schedulers can support fairness with negligible performance overheads.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2012R1A1A1014586 and No. 2013R1A2A2A01015514)

8. REFERENCES

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[2] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg. Fabscalar: composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA)*, 2011.

[3] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th annual International Symposium on Computer architecture (ISCA)*, 2011.

[4] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. *ARM whitepaper*, 2011.

[5] S. Herbert and D. Marculescu. Variation-aware dynamic voltage/frequency scaling. In *Proceedings of the 15th IEEE international symposium on High Performance Computer Architecture (HPCA)*, 2009.

[6] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the 17th*

international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

[7] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Utility-based acceleration of multithreaded applications on asymmetric CMPs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[8] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, 2010.

[9] 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 *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[10] 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 *Proceedings of the 31st annual International Symposium on Computer architecture (ISCA)*, 2004.

[11] Y. Kwon, C. Kim, S. Maeng, and J. Huh. Virtualizing performance asymmetric multi-core systems. In *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA)*, 2011.

[12] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *Proceedings of the 2009 Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[13] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[14] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2007.

[15] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proceedings of the 16th IEEE international symposium on High Performance Computer Architecture (HPCA)*, 2010.

[16] J. C. Saez, A. Fedorova, M. Prieto, and H. Vegas. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. In *Proceedings of the 7th ACM international conference on Computing Frontiers (CF)*, 2010.

[17] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, 2010.

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

[19] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[20] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proceedings of the 35th annual International Symposium on Computer Architecture (ISCA)*, 2008.

[21] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *Proceedings of the 22nd international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[22] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proceedings of the 39th annual International Symposium on Computer Architecture (ISCA)*, 2012.