Fairness-Aware Scheduling on Single-ISA Heterogeneous Multi-Cores

Kenzo Van Craeynest[†°] Shoaib Akram[†] Wim Heirman[†°] Aamer Jaleel[‡] Lieven Eeckhout[†]
[†]Ghent University, Belgium [°]ExaScience Lab, Belgium [‡]VSSAD, Intel Corporation

***Abstract*—Single-ISA heterogeneous multi-cores consisting of small (e.g., in-order) and big (e.g., out-of-order) cores dramatically improve energy- and power-efficiency by scheduling workloads on the most appropriate core type. A significant body of recent work has focused on improving system throughput through scheduling. However, none of the prior work has looked into fairness. Yet, guaranteeing that all threads make equal progress on heterogeneous multi-cores is of utmost importance for both multi-threaded and multi-program workloads to improve performance and quality-of-service. Furthermore, modern operating systems affinitize workloads to cores (pinned scheduling) which dramatically affects fairness on heterogeneous multi-cores.**

In this paper, we propose fairness-aware scheduling for single-ISA heterogeneous multi-cores, and explore two flavors for doing so. Equal-time scheduling runs each thread or workload on each core type for an equal fraction of the time, whereas equal-progress scheduling strives at getting equal amounts of work done on each core type. Our experimental results demonstrate an average 14% (and up to 25%) performance improvement over pinned scheduling through fairness-aware scheduling for homogeneous multi-threaded workloads; equal-progress scheduling improves performance by 32% on average for heterogeneous multi-threaded workloads. Further, we report dramatic improvements in fairness over prior scheduling proposals for multi-program workloads, while achieving system throughput comparable to throughput-optimized scheduling, and an average 21% improvement in throughput over pinned scheduling.

***Keywords*—heterogeneous multi-core, fairness-aware scheduling**

I. Introduction

Heterogeneous multi-cores can enable higher performance within a given power budget, or reduced power and energy consumption within a given performance target, by executing workloads on the most appropriate core type. Recent work has demonstrated the potential of heterogeneous multi-cores to dramatically improve processor energy- and power-efficiency [2], [6], [11], [15], [16], [17], [18], [19], [29]. Single-ISA heterogeneous multi-cores feature different core types while implementing the same instruction-set architecture (ISA). Existing commercial single-ISA heterogeneous multi-cores include NVidia's Kal-El [24] which integrates four performance-tuned cores along with one energy-tuned core, and ARM's big.LITTLE design [12], which enables a high-performance big core with a low-energy small core. For a given power budget, both these commercial offerings enable varying number of big and small cores to be simultaneously active. Doing so enables executing multiple workloads concurrently or allows a multi-threaded workloads to expose as many threads as there are cores in the system.

How to best schedule workloads (or threads) on the most appropriate core type is foundational to single-ISA heterogeneous multi-core processor. Making wrong scheduling decisions can lead to suboptimal performance and/or excess energy and power consumption. A significant body of work has focused on this scheduling problem for multi-program workloads. Some prior work uses workload memory intensity as an indicator to guide application scheduling [2], [6], [11], [15], [20], [29]. Others make offline scheduling decisions based on profiling information [6], [29], or use sampling-based solutions [2], [16], [17]. Model-based scheduling leverages analytical models to steer scheduling in order to overcome inaccuracies due to heuristics (e.g., memory dominance) and scalablity limits due to sampling [21], [31], [33]. Prior work on scheduling multi-threaded workloads on heterogeneous multi-cores has primarily focused on accelerating the serial fraction of code [1] and critical sections [32], and on identifying and accelerating critical bottlenecks [13] and threads using synchronization behavior [7].

All of the prior work in scheduling heterogeneous multi-cores focused on optimizing total system throughput. None of the prior work considered fairness as an optimization target. Yet, fairness, or guaranteeing that all threads and/or programs make equal progress, is of great importance. For example, in a barrier-synchronized multi-threaded workload, a thread that gets to run on a big core will just wait stalling on the barrier until all other threads running on the small cores have reached the barrier — yielding no performance benefit from heterogeneity. Guaranteeing fairness, or making sure all threads make equal progress, will lead to a more balanced execution, thereby improving overall application performance. For multi-program workloads, fairness is of utmost importance when it comes to system-level priorities and quality-of-service (QoS). In particular, system software (e.g., the operating system or the virtual machine monitor) essentially assumes all threads (or programs) make equal progress when run on the hardware. Yet, a thread/program that runs on a big core gets more work done than when run on a small core.

Leveraging existing multi-core schedulers on heterogeneous multi-cores does not provide fairness either. Schedulers in modern operating systems affinitize or pin threads or jobs to cores in order to minimize overhead of context switching and increase data locality [14]. We refer to this scheduling policy as pinned scheduling throughout the paper. With pinned scheduling, a thread pinned to a big core will make faster

progress compared to threads pinned to small cores, potentially leading to poor fairness and reduced performance. A straightforward approach to improve over pinned scheduling would be to allow threads take turns on the slow and fast cores. Blindly rotating threads between big and small cores can unnecessarily waste power and hurt performance due to the effects of context switches especially for threads that have similar performance on both core types. This suggests the need for an intelligent scheduling policy that improves fairness by only moving threads that are suffering from 'unfairness'.

In this paper, we propose fairness-aware scheduling for single-ISA heterogeneous multi-cores. We consider a number of mechanisms for achieving fairness. *Equal-time* scheduling strives at scheduling all threads onto a big core for an equal amount of time. Because equal time does not necessarily lead to equal progress, especially for heterogeneous workloads in which threads exhibit different execution behavior, we also propose *equal-progress* scheduling which strives at getting all threads to make equal progress. We consider three different ways for estimating progress: sampling, history and model-based estimations. Finally, we also explore tunable scheduling policies that trade off system throughput versus fairness. All of these scheduling strategies monitor a thread's progress or time during run-time, and dynamically reschedule threads to improve fairness. Fairness-aware scheduling not only improves fairness over pinned scheduling, it also improves system throughput by enabling threads to run on a big core type for some fraction of time. Further, it achieves a level of system throughput that is comparable to throughput-optimized scheduling as proposed in prior work, while dramatically improving fairness.

By proposing fairness-aware scheduling, we are addressing the challenge of how to best run multi-threaded workloads (and multi-program workloads with QoS constraints) on a single-ISA heterogeneous multi-core. This is obviously not a concern when running a single program on a single heterogeneous sub-system, for which the user can discern background tasks from high-performance tasks — the suggested usage model of the ARM big.LITTLE system [12]. The challenge pops up when running multi-threaded workloads on a chip with multiple such heterogeneous sub-systems, as is the case for Samsung's Galaxy S4 processor (Exynos 5 Octa, four big.LITTLE sub-systems on a single chip). Running one (or some) of the threads on a big core and the others on a small core is going to maximize performance for a given power budget, provided that a fairness-aware scheduling mechanism is in place to make sure all threads make equal progress and equally benefit from the big core. By doing so, fairness-aware scheduling not only improves performance, it also reduces energy consumption.

Our experimental evaluation includes both multi-threaded and multi-program workloads across a range of heterogeneous multi-core architectures. We report average performance improvements of 14% (and up to 25%) for the multi-threaded workloads through fairness-aware scheduling. Equal-progress scheduling improves performance by 32% on average for heterogeneous multi-threaded workloads; equal-time and equal-progress scheduling perform equally well on homogeneous multi-threaded workloads in which all threads run the same code. For multi-program workloads on a heterogeneous multi-core with one big and three small cores, fairness-aware scheduling achieves an average fairness level of 86%, a significant improvement over pinned and throughput-optimized scheduling with fairness levels of 56% and 64%, respectively. Moreover, fairness-aware scheduling improves system throughput by 21% on average over pinned scheduling, while being within 3.6% on average compared to throughput-optimized scheduling. Scheduling that trades off fairness for throughput enables reducing the maximum throughput reduction compared to throughput-optimized scheduling while achieving similar levels of fairness compared to fairness-aware scheduling. Overall, these results demonstrate that fairness-aware scheduling is key to optimizing performance on single-ISA heterogeneous multi-cores for both multi-threaded and multi-program workloads.

II. Motivation

Before elaborating on fairness-aware scheduling, we now motivate the need for fairness for both multi-threaded and multi-program workloads on heterogeneous multi-cores.

A. Fairness

We first define fairness for heterogeneous multi-cores, along the lines of prior definitions of fairness in multi-threaded and multi-core systems [8], [10]. We denote T_{het} as the number of cycles to execute a thread on the heterogeneous multi-core when run simultaneously with other threads or applications; T_{big} is defined as the time it takes to execute on the big core (of the same heterogeneous multi-core) when run in isolation. The slowdown of thread i on the heterogeneous multi-core is then defined as the slowdown when running on the heterogeneous multi-core compared to running on the big core in isolation:

$$S_i = \frac{T_{het,i}}{T_{big,i}}. \tag{1}$$

We define a schedule to be fair if the slowdowns of all (equal-priority) threads running simultaneously on the heterogeneous multi-core are the same, similarly to prior work on fairness [8], [10]. A frequently used metric for fairness is to compute the ratio of the minimum versus maximum slowdowns among all simultaneously running threads. One problem with this metric is that it only considers the outlier threads, and does not take into account the 'average' thread. We therefore propose and use a different metric in our work, which is based on the statistically well-founded coefficient of variation:

$$fairness = 1 - \frac{\sigma_S}{\mu_S}. \tag{2}$$

μ_S is the average slowdown across all threads, and σ_S is the standard deviation across all slowdowns of all threads. The fraction σ_S/μ_S is the so-called coefficient of variation and measures the variability in slowdown in relation to the mean slowdown — hence, it is a measure for the unfairness, i.e., the larger the variability in slowdown, the more unfair the execution is. One minus the coefficient of variation then is a measure for fairness. Fairness is a higher-is-better metric, and a fairness of one means that all threads make equal progress, relative to running on the big core in isolation.

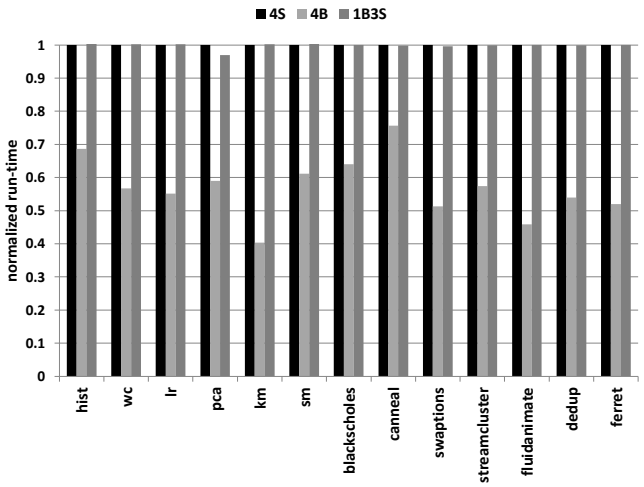

Figure 1. Normalized run-time on a homogeneous multi-core with 4 small cores (4S), 4 big cores (4B), and a heterogeneous multi-core with one big and three small cores (1B3S) while keeping threads pinned to cores.

B. Multi-threaded workloads

In order to illustrate the importance of achieving fairness, we first consider a number of multi-threaded workloads from the Phoenix [28] and PARSEC [4] benchmark suites, and run these workloads on a heterogeneous multi-core with one big and three small cores (1B3S); the last-level cache is shared among all four cores. (We refer to later for a detailed description of the experimental setup.) We pin each thread to a core, and compare heterogeneous multi-core performance against homogeneous multi-cores with four big (4B) versus four small cores (4S), see Figure 1. A homogeneous multi-core with big cores achieves a speedup ranging between 1.25× to 2.5× (run-time reduction by 20% to 60%) compared to a homogeneous multi-core with small cores. This is to be expected given the relative performance difference between big and small cores.

The more interesting result is that a heterogeneous multi-core achieves no speedup over a homogeneous multi-core with all small cores for most of the benchmarks. The reason is that for barrier-synchronized multi-threaded workloads, the threads pinned onto a small core determine overall application performance, i.e., all other threads have to wait for the threads running on the small cores due to synchronization (i.e., barriers). Thus, even though the thread that runs on the big core makes faster progress compared to the threads running on the small cores, there is no performance improvement compared to an all small core system. The situation is different for work-stealing type of multi-threaded workloads where the thread running on the big core 'steals' work from the small cores. As such, fairness is not an important criterion for work-stealing type of multi-threaded workloads.

Thus, for barrier-synchronized multi-threaded workloads, guaranteeing fairness improves overall performance. Fairness is accomplished by making sure that all threads get an equal amount of work done, which enables all threads to reach the barriers at the same time.

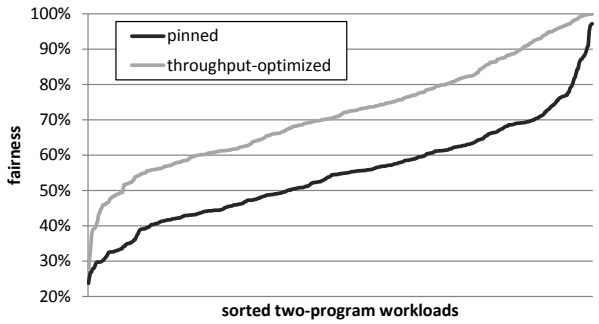

Figure 2. Fairness for a 1B1S system for pinned versus throughput-optimized scheduling using PIE for 500 randomly chosen two-job mixes.

C. Multi-program workloads

Figure 2 quantifies fairness for a heterogeneous multi-core with one big and one small core (1B1S) for both pinned scheduling and throughput-optimized scheduling while running multi-program workloads composed of random mixes of SPEC CPU2006 benchmarks. While random mixes of SPEC CPU2006 workloads may not be the standard use case of commercial heterogeneous processors, we use them as a substitute for emulating the behavior of random tasks concurrently executing on a heterogeneous multi-core. The figure shows that pinned scheduling causes some programs to make poor progress, i.e., the program that runs on the small core makes less progress than the program that runs on the big core. Overall, fairness through pinned scheduling equals 55% on average and is as low as 24% for some workload mixes (see bottom left of the curve in Figure 2). State-of-the-art throughput-optimized scheduling using PIE [33] not only improves system throughput by 26.6% on average, it also improves fairness by a significant margin from 55% to 72% on average. The reason is that throughput-optimized scheduling reschedules programs during run-time to improve throughput, and by dynamically migrating programs between big and small cores in response to time-varying execution behavior, it also improves fairness. Yet, fairness is fairly low: 72% on average, and as low as 27% (see bottom left in Figure 2). In other words, both pinned scheduling and throughput-optimized scheduling are largely unfair which may compromise quality-of-service.

III. Fairness-Aware Scheduling

Having motivated the importance of fairness as an optimization criterion on single-ISA heterogeneous multi-cores, we now propose equal-time and equal-progress fairness-aware scheduling in the next two subsections.

A. Equal-time scheduling

Equal-time scheduling strives at achieving fairness by running each thread on each core type for an equal amount of time. This is done by keeping track of how often (for how many time slices) a thread has run on all core types, and reschedule if necessary to make sure all threads have run on either core type for an equal number of time slices. Round-robin or random selection of a thread that runs on a small core to next run on the big core is an implementation of equal-time

scheduling. Note we do not migrate threads among cores of the same type in order to preserve data locality.

A pitfall with equal-time scheduling is that spending equal time on either core type does not necessarily imply fairness. Some threads experience a larger slowdown from running on a small core than others — these threads get proportionally less work done when scheduled on a small core. Hence, although all threads spend equal time on either core type, threads that experience higher slowdowns on the small cores, will make proportionally less progress. This leads to an unfair system. We therefore propose equal-progress scheduling in the next section which strives at getting equal work done on either core type, and by consequence achieve equal progress.

Note that when all threads exhibit the same (or similar) execution behavior — a so-called homogeneous workload — equal-progress scheduling is in fact identical to equal-time scheduling. Because of the one-to-one relationship between time and work done, i.e., equal time leads to equal work, scheduling all threads on either core type for equal amounts of time leads to equal amounts of work done on either core type. This is not the case for heterogeneous workloads in which threads execute different codes (and are therefore heterogeneous by design), as we will demonstrate later in this paper. Similarly, homogeneous-by-design workloads for which different threads end up processing different parts of the input data may exhibit heterogeneous behavior, and may therefore benefit from equal-progress scheduling over equal-time scheduling.

B. Equal-progress scheduling

Equal-progress scheduling strives at getting all threads to make equal progress. Or, in other words, it strives at making sure all threads experience equal slowdown, per the definition of fairness (Equation 2). Equal-progress scheduling continuously monitors fairness and dynamically adjusts the scheduling to achieve fairness. This involves computing the slowdowns for all threads and scheduling the thread with the currently highest slowdown on the big core. (If there are multiple big cores in the system, the threads with the top-n highest slowdowns are scheduled on a big core.) Scheduling the thread with the currently highest slowdown on a big core will reduce its slowdown compared to the other threads (which are scheduled on a small core). As a result, the threads' slowdowns will converge and fairness is achieved.

Computing slowdowns for all threads is where the key challenge lies for equal-progress scheduling. In order to compute a thread's slowdown, we need to know the total execution time on the heterogeneous multi-core as well as the total execution time if we were to execute the thread on a big core in isolation, see Equation 1. The former, total execution time on the heterogeneous multi-core, is readily available by counting the number of time slices TS_i the thread has been running so far (on both core types). The latter, total execution time on the big core in isolation, is not readily available and needs to be estimated during run-time. We estimate the isolated, big core execution time by counting the number of time slices the thread was run on the big versus small cores, and by rescaling the time run on the small cores with an estimated big-versus-small-core scaling factor R. A thread's slowdown is then computed as follows:

$$S_i = \frac{T_{het,i}}{T_{big,i}} = \frac{TS_{big,i} + TS_{small,i}}{TS_{big,i} + TS_{small,i}/R_i}. \quad (3)$$

This formula can be trivially extended to heterogeneous multi-cores with more than two core types.

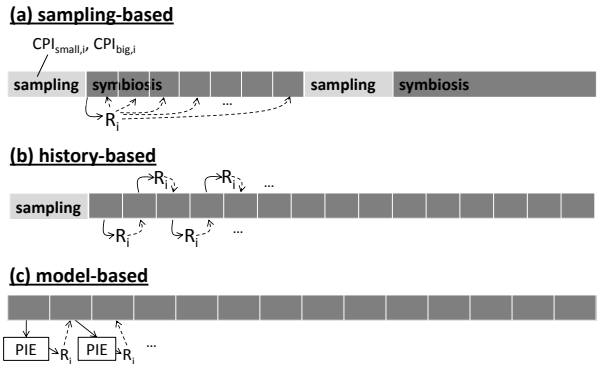

Figure 3. Equal-progress scheduling: sampling-based, history-based and model-based.

We explore three methods for estimating the big-versus-small-core scaling factor R, see also Figure 3.

- *Sampling-based* scheduling considers a sampling and symbiosis phase. During the sampling phase, the scheduler maps each thread at least once on each core type, and computes the scaling factor R as the ratio between the CPI on the small versus big core:

  $$R = CPI_{small}/CPI_{big}.$$

  The scheduler then uses the computed R factor during the symbiosis phase. We assume the symbiosis phase is ten times longer than the sampling phase in our setup.

- *History-based* scheduling computes the CPI seen on both small and big cores and uses the ratio for estimating slowdown. The benefit over sampling-based scheduling is that history-based scheduling continuously adjusts the computed big-to-small ratio based on the most recent CPI values. In order for history-based scheduling to work in practice, it needs a bootstrap phase in which all threads get to run on all core types at least once.

- *Model-based* scheduling continuously monitors CPI on either core type and estimates the big-to-small-core ratio using an analytical model. The key benefit is that model-based scheduling continuously updates the big-to-small-core ratio based on the most recent time slice. Sampling- and history-based scheduling on the other hand use stale CPI values to compute the ratio. We use the PIE model [33] for estimating the big-to-small-core ratio. The effectiveness of the model-based approach depends on the accuracy of the model, and may require hardware support for computing the inputs for the model (as is required for PIE). The sampling-based and history-based methods on the other hand do not require hardware support, and they use real performance measurements, which may

be more accurate than model estimates, albeit being stale.

C. Trading fairness for throughput

So far, we considered fairness as the only optimization criterion, i.e., the proposed fairness-aware scheduling mechanisms strive at achieving fairness and are oblivious to system throughput. However, in some practical use cases, fairness is not the only optimization criterion, and system throughput is at least equally important. For example, in a batch-style, throughput-oriented system, maximizing system throughput might be of primary importance, and fairness among users or jobs might be a secondary concern. Hence, it might make sense to provide a flexible scheduling algorithm that enables trading off fairness for throughput, and vice versa.

We therefore propose a scheduling approach that trades off fairness and system throughput: *Guaranteed-fairness* scheduling optimizes for system throughput, yet when fairness drops below a given threshold $\theta_{fairness}$, scheduling defers to optimizing fairness until fairness reaches at least the threshold, after which it defers again to throughput-optimized scheduling. Guaranteed-fairness scheduling thus needs to continuously monitor slowdowns and estimate fairness, as done for fairness-aware scheduling. We consider different fairness thresholds for guaranteed-fairness scheduling in the evaluation section of this paper.

D. Rescheduling granularity

The fairness-aware scheduling algorithms proposed in this paper dynamically reschedule threads across core types during run-time. This is done at the granularity of a time slice. There are a number of factors that affect a good choice of time slice granularity. A small time slice potentially makes the system more responsive, i.e., the scheduling algorithm can guarantee fairness at smaller time scales and more quickly react to time-varying execution behavior. On the other hand, a small time slice also incurs more migration overhead when threads are more frequently rescheduled. The migration overhead not only includes overhead due to a context switch, it also incurs overhead for warming hardware state, especially in the memory hierarchy. Whereas context switch overhead incurs a fixed cost for restoring architecture state, the overhead for warming hardware state depends on the workload and its working set size, as well as the memory hierarchy.

Van Craeynest et al. [33] did an extensive evaluation to quantify migration overhead for both shared and private last-level caches (LLCs) as a function of time slice granularity. They found the migration overhead to be less than 1.5% across all workloads for a 4 MB shared LLC for a 1 ms time slice, and less than 0.6% for a 2.5 ms time slice. They also explored migration overhead for private LLCs and found the overhead to be small as well (although slightly higher compared to shared caches) because the cache coherency protocol can get the data from another core's private LLC instead of memory. Our own experimental evaluation confirms these findings, and we consider a 1 ms time slice unless mentioned otherwise.

E. Hardware vs. software scheduling

Implementing fairness-aware scheduling can be done both in hardware and in software. When implemented in hardware, the fairness-aware scheduling would need a small time slice, e.g., 1 ms, while system software (the OS or VMM) uses a larger time slice, e.g., 4+ ms. By doing so, the hardware would be able to provide the abstraction to software of homogeneous hardware, while dynamically rescheduling threads among the cores in a heterogeneous multi-core within an OS time slice. For example, a heterogeneous multi-core with one big and three small cores, may then be exposed to software as a homogeneous multi-core with four cores and a 4 ms time slice; the hardware however, would then dynamically reschedule threads among the cores at a 1 ms time slice. By scheduling for fairness, hardware would expose itself as a homogeneous multi-core in which all threads make equal progress. System software does not need to be changed and is oblivious to the heterogeneity in hardware. In contrast, implementing fairness-aware scheduling in software requires modifications to the OS or VMM to keep track of each thread's progress, and enables guaranteeing fairness at a larger time scale. Fairness-aware scheduling can be implemented in both software and hardware, and works at different time scales, as we demonstrate later in the paper.

IV. Experimental Setup

Before describing and analyzing results, we first describe our experimental setup.

A. Simulated architectures

We use Sniper [5] for conducting the simulation experiments in this paper. Sniper is a parallel, hardware-validated, x86-64 multi-core simulator capable of running both multi-program and multi-threaded applications. We configure Sniper to model heterogeneous multi-core processors with big and small cores. The big core is a 4-wide out-of-order processor core; the small core is a 4-wide (stall-on-use) in-order processor core. We assume both cores run at a 2.6 GHz clock frequency. Further, we assume a cache hierarchy with separate 32 KB L1 instruction and data caches, and a 256 KB L2 cache; we assume the L1 and L2 caches to be private per core. The L3 last-level cache (LLC) is shared among all cores, for a total size of 16 MB. We consider the LRU replacement policy in all of the caches.

As mentioned before, the time slice granularity is set to be 1 ms, in order to be able to exploit time-varying workload execution behavior while keeping migration overhead small. The overhead for migrating a workload from one core to another has three components. A fixed 1,000 cycle penalty for storing and restoring the architecture state (at most a few kilobytes of state). Additionally, we also model the overhead due to the time it takes to drain a core's pipeline prior to migration. Finally, we account for the migration overhead due to cache effects. The latter is by far the largest and most important component, being at least two order of magnitude larger than the first two components combined.

Table I. MULTI-THREADED BENCHMARKS USED IN THIS STUDY.

Suite	*Benchmark*	*Input*
PARSEC	blackscholes	simmedium
	canneal	simmedium
	swaptions	simmedium
	streamcluster	simmedium
	fluidanimate	simmedium
	dedup	simmedium
	ferret	simmedium
MapReduce	histogram	1.5 GB image
	word count	100 MB file
	linear regression	100 MB file
	PCA	1024 x 1024 matrix
	K-means	64 clusters, 65536 points 256-dimension vectors
	string match	100 MB file

B. Workloads

We consider both multi-program and multi-threaded workloads in our experiments. The multi-program workloads are composed out of SPEC CPU2006 benchmarks; there are 26 benchmarks in total, which along with all of their reference inputs leads to 55 benchmarks in total. We select representative simulation points of 750 million instructions each; these simulation points were selected using PinPoints [25]. When running multi-program workloads, we stop the simulation as soon as the first benchmark in the workload mix reaches the end of its simulation point; this corresponds to hundreds of time slices. We quantify system throughput using the STP metric [8] (also called weighted speedup [30]) which quantifies the aggregate throughput achieved by all cores in the system. We use Equation 2 when reporting fairness. We consider 500 randomly chosen two-job workload mixes, and 200 randomly chosen four-job and eight-job mixes.

The multi-threaded benchmarks considered in this paper are selected from Phoenix [28] and PARSEC [4], see Table I. The Phoenix benchmarks are MapReduce workloads with Map, Reduce and Merge phases, using the Metis [22] library for shared-memory multi-core processors. These workloads are homogeneous (i.e., all threads run the same code) and barrier-synchronized between parallel phases. Most of the PARSEC benchmarks are homogeneous and barrier-synchronized as well, expect for `dedup` and `ferret` which are pipelined programs. The latter two benchmarks are therefore heterogeneous, i.e., different threads execute different codes and communicate through a producer-consumer relationship. We use the `simmedium` inputs for PARSEC. The run-times for the multi-threaded benchmarks are such that we simulate several hundreds upto a couple thousands of time slices. We run the benchmarks to completion and measure total run-times.

V. EVALUATION

We now evaluate fairness-aware scheduling and compare against two alternative scheduling policies, namely throughput-optimized and pinned scheduling. Throughput-optimized scheduling is state-of-art dynamic PIE scheduling [33] which uses a simple analytical model to predict on which core type to map which program or thread in order to optimize system throughput. PIE dynamically reschedules threads to exploit time-varying execution behavior. Pinned scheduling is our baseline, and reflects current practice in contemporary operating system schedulers, as done in the Linux 2.6 kernel [14]. Pinned scheduling maps threads to cores and keeps threads pinned to cores in order to improve data locality and affinity. When reporting performance numbers for pinned scheduling, we consider multiple random mappings of threads to cores and report average performance across those random mappings. We believe pinned scheduling is a reasonable baseline to compare against. In case fairness-aware scheduling were implemented in hardware, pinned scheduling reflects system software (randomly) mapping and pinning threads to virtual cores, while hardware scheduling reschedules threads to physical cores to optimize fairness. In case fairness-aware scheduling were implemented in software, pinned scheduling reflects optimizing for data locality.

A. Multi-program workloads

We first evaluate fairness-aware scheduling in the context of multi-program workloads.

1) Equal-time versus equal-progress: Figure 4 reports throughput and fairness for both equal-time and equal-progress fairness-aware scheduling, compared to pinned scheduling and throughput-optimized scheduling, for a 1B1S heterogeneous multi-core with one big and one small core. We consider history-based equal-progress scheduling here, and evaluate other equal-progress policies later. The workloads on the horizontal axis are sorted. Pinned scheduling performs the worst in terms of fairness, with an average fairness of 55%. Throughput-optimized scheduling improves fairness somewhat to 72% on average. By dynamically rescheduling threads among cores, throughput-optimized scheduling not only improves throughput, it also improves fairness; a thread may get to run on either core type for some fraction of time. Fairness-aware scheduling achieves the highest fairness (92% on average), and equal-progress scheduling slightly outperforms equal-time scheduling. The highest unfairness observed across all the 500 job mixes is no higher than 38%, which is a substantial improvement over both pinned and throughput-optimized scheduling with unfairness numbers up to 73%. Fairness-aware scheduling results in a slightly lower system throughput compared to throughput-optimized scheduling: 22.0% and 21.9% for equal-progress and equal-time scheduling versus 26.6% for throughput-optimized scheduling. The reason why equal-progress scheduling outperforms equal-time scheduling in terms of throughput is that it takes into account the amount of work done on either core type, and not just time.

2) Scalability: We now evaluate fairness-aware scheduling as a function of the number of cores and different ratios of big to small cores. Figure 5 shows average fairness as well as throughput values for 1B1S, 1B3S, 3B1S, 1B7S and 7B1S systems. The overall conclusion is that fairness-aware scheduling achieves the highest fairness across the board, with average fairness values ranging between 79 and 92%, which is significantly higher compared to pinned and throughput-optimized scheduling with average fairness values around 50 and 70%, respectively. Equal-progress scheduling achieves higher fairness compared to equal-time scheduling for the 1B3S and 1B7S systems, but achieves similar average levels of fairness for the other systems. The reason is that equal-progress scheduling computes slowdowns based on actual progress —

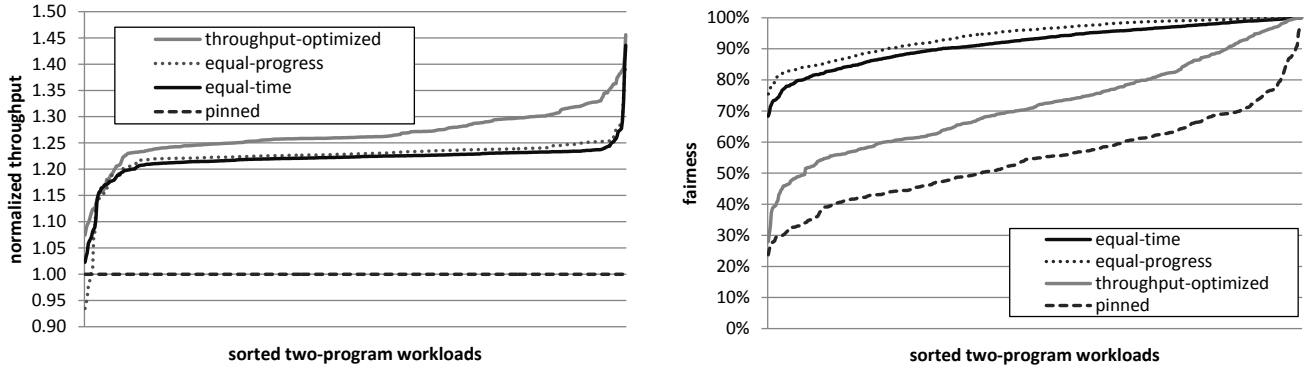

Figure 4. Comparing scheduling algorithms relative to pinned scheduling in terms of throughput (left graph) and fairness (right graph) for a 1B1S heterogeneous multi-core with one big and one small core.

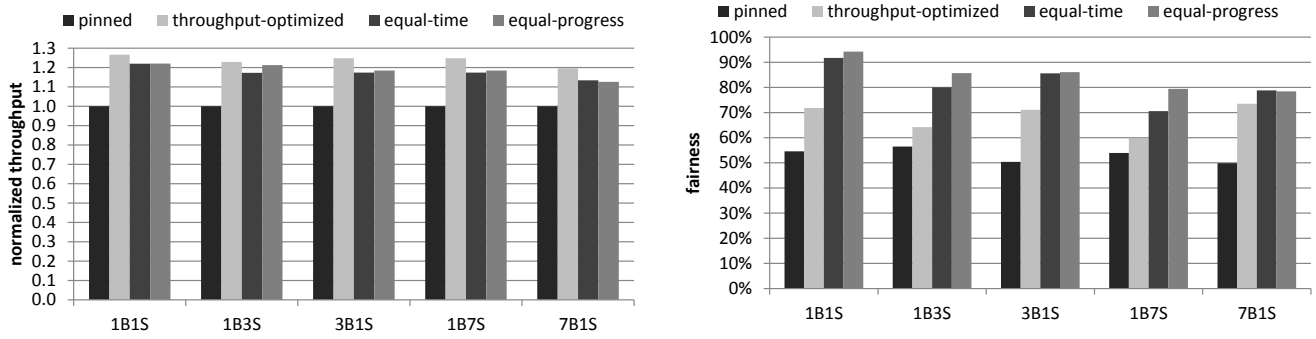

Figure 5. Fairness-aware scheduling as a function of core count in terms of throughput (left graph) and fairness (right graph).

not time — which is more accurate and turns out to be more critical when the number of big cores is small compared to the number of small cores. In other words, as the relative number of big cores decreases, the big core becomes a bottleneck and making accurate slowdown predictions becomes more critical towards optimizing fairness.

Note also that fairness degrades with decreasing relative fractions of big cores, even under fairness-aware scheduling. Fairness degrades from 92% for a 1B1S system (1/2 the cores are big cores), to 79% for a 1B7S system (1/8th the cores are big cores). This can be understood intuitively because the big core is increasingly becoming a bottleneck as the relative number of big cores decreases in the system. In other words, fairness is easier to be achieved when the number of big versus small cores is more balanced.

3) Equal-progress scheduling: As mentioned earlier in the paper, there are a number of ways for estimating the big-to-small-core scaling ratio in equal-progress scheduling. We considered the history-based method so far; we now evaluate the other two, sampling- and model-based, methods. Figure 6 compares these three methods in terms of throughput and fairness for a 1B7S system. Sampling-based scheduling performs worst (both in terms of fairness and throughput) because it periodically estimates the big-to-small-core scaling ratio, after which the scaling ratio is used during the symbiosis phase. Sampling incurs overhead and is unable to quickly adapt to time-varying workload behavior. Note sampling-based scheduling performs even worse compared to equal-time scheduling. The history-based and model-based methods perform much better, and both outperform sampling-based

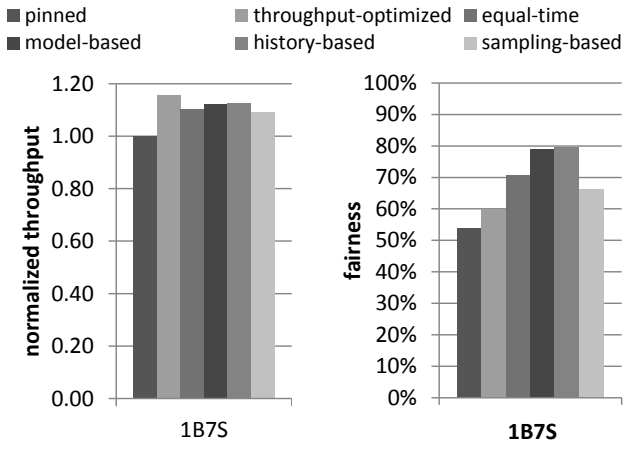

Figure 6. Evaluating different methods for estimating the big-to-small-core scaling factor in equal-progress scheduling for a 1B7S system.

scheduling and equal-time scheduling, as they continuously update the big-to-small-core scaling ratio.

We find history-based scheduling to typically outperform model-based scheduling, albeit by a small margin. As mentioned before, model-based scheduling does not rely on stale data to compute the big-to-small-core ratio but is limited by the accuracy of the underlying model; history-based scheduling on the other hand computes the big-to-small ratio based on real hardware measurements instead of a model, which might provide more accurate big-to-small-core scaling ratios in

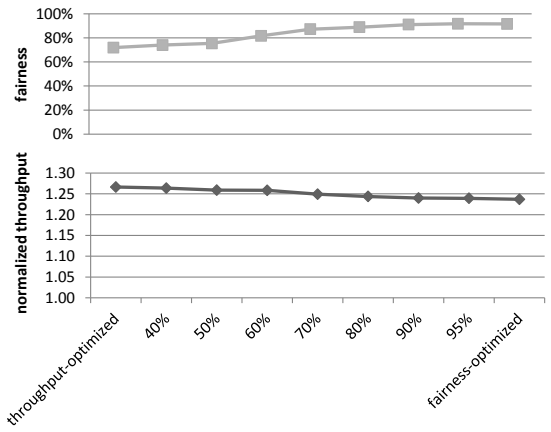

Figure 7. Trade-off between fairness and throughput-optimized scheduling for a 1B1S system.

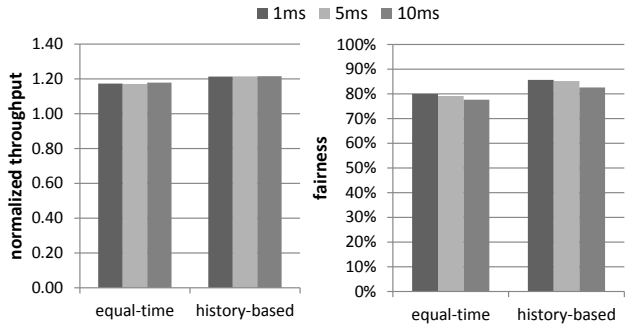

Figure 8. System throughput and fairness for equal-time and equal-progress (history-based) scheduling as a function of time slice granularity.

case the hardware measurements are fairly recent. Moreover, optimizing for fairness enforces threads to migrate across core types, which enables the history-based approach to continuously update the big-to-small-core ratio using fairly recent performance numbers on both the small and big cores.

4) Trading fairness vs. throughput: As mentioned earlier in the paper, optimizing system performance is often a complex trade-off in terms of system throughput (i.e., getting as much jobs done per unit of time) versus user-level experience (i.e., all users should be treated in a fair way). The throughput-optimized and fairness-aware scheduling policies optimize system throughput and fairness, respectively, and are completely oblivious to the other optimization criterion. We now evaluate guaranteed-fairness scheduling which optimizes system throughput unless fairness drops below a given threshold, after which it defers to fairness-aware scheduling; once fairness is above the threshold, it optimizes system throughput again. Figure 7 evaluates guaranteed-fairness scheduling in terms of system throughput and fairness. This graph illustrates that turning the threshold 'knob' enables trading off fairness for throughput and vice versa. Setting the threshold to a higher value leads to high fairness, alike fairness-aware scheduling. Setting the threshold to a lower value leads to high levels of system throughput, alike throughput-optimized scheduling.

5) Time slice granularity: So far, we assumed a 1 ms time slice. Figure 8 evaluates fairness-aware scheduling across different time slices, including 1, 5 and 10 ms, for a 1B3S system. The motivation for exploring larger time slices is to evaluate fairness-aware scheduling in system software; smaller time slices correspond to implementing fairness-aware scheduling in hardware so that system software is oblivious to hardware heterogeneity, as previously discussed. We conclude from Figure 8 that both equal-time and equal-progress scheduling are (largely) insensitive to time slice granularity, i.e., similarly high levels of system throughput and fairness are achieved across different time slices. Fairness only slightly decreases with increasing time slices, the reason being that fairness converges slower with larger time slices; given the fixed workload (and run-time) this leads to slightly lower fairness values.

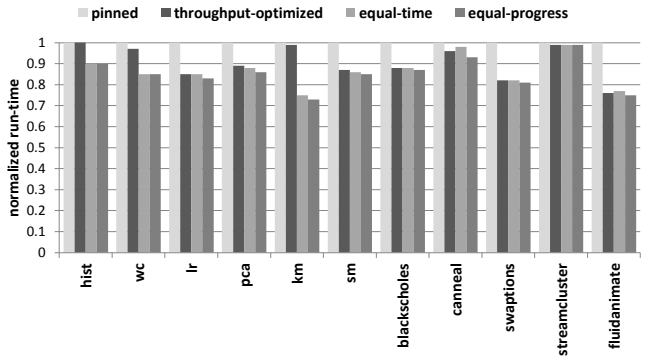

Figure 9. Comparing scheduling algorithms relative to pinned scheduling for a 1B3S system running homogeneous multi-threaded applications.

B. Multi-threaded workloads

We now evaluate fairness-aware scheduling for multi-threaded applications.

1) Homogeneous workloads: We first consider all the MapReduce workloads as well as the homogeneous workloads from the PARSEC benchmark suite. Figure 9 compares the various scheduling policies for a 1B3S heterogeneous multi-core system in terms of execution time normalized to pinned scheduling. The key result from this graph is that fairness-aware scheduling improves execution time by 14% on average and up to 25% over pinned scheduling. Interestingly, equal-time and equal-progress scheduling perform equally well. The intuitive understanding is that these workloads are homogeneous (all threads execute the same code and exhibit the same execution behavior), and enforcing equal time therefore leads to enforcing equal progress as well. Fairness-aware scheduling forces all threads to make equal progress by running on the big core for an equal share. This eventually leads to all threads reaching the barriers at roughly the same time. Under pinned scheduling on the other hand, the one thread that gets scheduled onto the big core reaches the barrier before the other threads; because this thread has to wait for the other threads on the small cores to reach the barrier, scheduling one of the thread on the big core does not contribute to overall performance, yielding no benefit from heterogeneity. By making sure all threads benefit from the big core, fairness-aware scheduling forces all threads to make equal progress, thereby reaching the barrier at the same time and improving overall run-time.

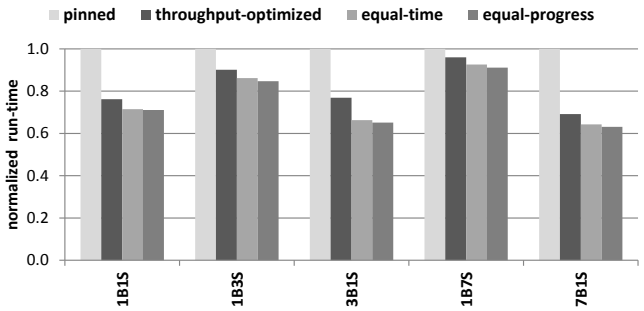

Figure 10. Fairness-aware scheduling for different heterogenous multi-core configurations for the homogeneous multi-threaded applications.

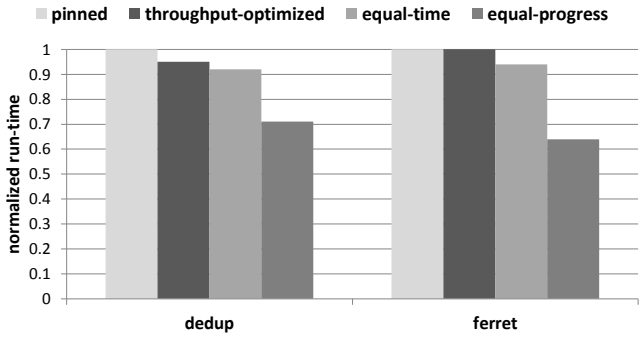

Figure 11. Comparing scheduling algorithms relative to pinned scheduling for a 1B3S system running heterogeneous multi-threaded applications.

Throughput-optimized scheduling improves performance for most benchmarks but not all, leading to an average improvement of 10% on average. The reason is that throughput-optimized scheduling improves fairness, as we have seen for the multi-program workloads, by migrating threads across core types while optimizing system throughput. However, the fact that fairness improves is a side-effect from optimizing for throughput; fairness-aware scheduling which specifically ensures that threads make equal progress leads to shorter runtimes.

Figure 10 reports average results across different heterogeneous multi-core configurations for the homogeneous multi-threaded workloads. The conclusion is essentially the same as what we reported earlier for the individual benchmarks. Fairness-aware scheduling improves performance substantially over pinned scheduling: we report average performance improvements ranging between 7.5% (for 1B7S) and 35% (for 7B1S). Further, equal-time and equal-progress scheduling perform equally well (the benefit from equal-progress scheduling over equal-time scheduling is marginal). Interestingly, performance improves with larger fractions of big cores in the system (compare 3B1S versus 1B3S, and 7B1S versus 1B7S) under fairness-aware scheduling. (In contrast, performance does not improve under pinned scheduling because the application has to wait for the slowest thread running on a small core anyways.) The reason is that fairness-aware scheduling gets to distribute and map threads across small and big cores, and when there are more big cores in the system, the average performance seen by all threads will be higher with more big cores in the system, leading to better overall performance.

2) Heterogeneous workloads: Figure 11 reports normalized execution time for fairness-aware scheduling for the two heterogeneous PARSEC benchmarks, `dedup` and `ferret`. The key conclusion from this graph is that although equal-time scheduling improves performance somewhat over pinned scheduling (8% for `dedup` and 6% for `ferret`), equal-progress scheduling improves performance by as much as 29% for `dedup` and 36% for `ferret`. The reason is that these workloads are heterogeneous, and, as a result, equal time does not necessarily correspond to equal progress. As the different threads execute different code and exhibit different execution behavior, they experience different big-to-small-core performance ratios and hence accounting for the different ratios is important for achieving fairness. Equal-progress scheduling does account for the fact that different threads make different progress and schedules threads such as to improve fairness, which ultimately leads to better overall application performance.

VI. Related Work

We now describe related work in scheduling heterogeneous multi-cores and homogeneous multi-cores with clock heterogeneity.

A. Scheduling heterogeneous multi-cores

There exists a fairly large body of prior work on scheduling for single-ISA heterogeneous multi-cores (with different core types) running multi-program workloads. Sampling-based scheduling runs each job in the mix on either core type to gauge the most energy-efficient core type [2], [17], [26]. A major limitation of sampling-based scheduling is that it scales poorly with increasing core count: an infrequent core type (e.g., a big core in a one-big, multiple-small core system) quickly becomes a bottleneck. Static scheduling [6], [29] does not suffer from scalability issues, however, it requires offline program analysis and/or profiling and is unable to adapt to input-sensitive and time-varying execution behavior. Memory-dominance scheduling [2], [6], [11], [15], [20], [29] schedules programs that exhibit frequent memory-related stalls on the small core and compute-intensive programs on the big core. Memory-dominance scheduling however, may lead to suboptimal scheduling when memory intensity alone is not a good indicator for workload-to-core mapping. Model-based scheduling [21], [31], [33] uses models to predict performance on other core types during run-time in order to dynamically schedule programs on the most performance-efficient core type, while taking into account both memory and compute intensity. Van Craeynest et al. [33] proposed the PIE model for heterogeneous multi-core with big out-of-order and small in-order cores. The PIE model uses simple analytical models to predict how core architecture affects exploitable ILP and MLP and its impact on CPI. Scheduling using the PIE model assumes hardware support for computing CPI stacks as well as a few model inputs. The hardware cost is limited to roughly 15 bytes of storage. Again, all of this prior work focused on optimizing system throughput while running multi-program workloads, and none looked into optimizing fairness nor multi-threaded application scheduling.

A handful of papers considered multi-threaded workloads and how to best schedule these on heterogeneous multi-core

processors. Annavaram et al. [1] mitigate Amdahl's Law by accelerating serial portions of a parallel workload. Suleman et al. [32] migrate threads executing likely to contend critical sections to a big core. Joao et al. [13] identify and accelerate most critical bottlenecks due to critical sections, barriers, pipeline stages. Du Bois et al. [7] propose a new metric to measure thread criticality based on synchronization behavior, which they use to accelerate the most critical thread(s). These prior works are orthogonal to our work. Whereas we focus on executing parallel code sections as efficiently as possible on heterogeneous multi-cores, this other prior work focuses on serial code sections and/or accelerating workload imbalance as a result of synchronization behavior.

B. Performance-asymmetric homogeneous multi-cores

A number of recent research papers address scheduling issues for performance-asymmetric homogeneous multi-cores in which different cores run at different clock frequencies, i.e., all cores implement the same microarchitecture but have a separate clock domain. None of these papers consider core microarchitecture diversity on the chip. Age-based scheduling [18] predicts the remaining execution time of a thread in a multi-threaded program and schedules the oldest thread on the big core. Bhattacharjee and Martonosi [3] predict critical threads in barrier-synchronized multi-threaded applications on homogeneous multi-core hardware, and scale down voltage and frequency of cores running non-critical threads in order to conserve power and energy. Li et al. [19] schedule multiple jobs on the high-frequency core and a single job on the low-frequency cores in order to improve fairness. Rangan et al. [27] explore throughput-optimizing and fairness-aware scheduling algorithms for homogeneous multi-cores for which each core runs at a different clock frequency due to within-die process variations. The proposed scheduling algorithms do not readily apply to heterogeneous multi-cores with core microarchitecture diversity; in addition, they consider multi-program workloads only. Michaud et al. [23] propose a scheduling algorithm for temperature-constrained multi-cores in which threads migrate between hot and cold cores in order to avoid hotspots while achieving fairness among threads. Fedorova et al. [9] propose a scheduling approach for multi-cores (with different cores running at different clock frequencies) that ensures that each thread's execution time is balanced across all cores; the key difference with the equal-time scheduling approach proposed in this paper is that we balance time across core types (not cores), and by doing so we avoid unnecessary migrating among cores of the same type.

VII. Conclusion

Current multi-core schedulers in modern operating systems affinitize or pin threads to cores, which leads to unfair performance on heterogeneous multi-cores. For barrier-synchronized multi-threaded workloads, unfair performance leads to thread(s) running on a big core to wait at barriers for the other threads running on the small cores, yielding no performance benefit from heterogeneity. For multi-program workloads, unfair performance may compromise quality-of-service because of large variability in performance across simultaneously running programs. Optimizing for system throughput, as proposed in a significant body of recent work, improves fairness somewhat by dynamically scheduling threads across core types during run-time while optimizing for throughput in response to time-varying workload behavior, yet, fairness is still poor.

This paper proposed fairness-aware scheduling which optimizes for fairness as its primary optimization objective. We described two techniques for making sure all threads get to run on either core types for equal shares. Equal-time scheduling schedules threads such that they all spend equal amounts of time on either core type. Equal-progress scheduling strives at getting all threads to make equal progress, and we described three methods for dynamically estimating a thread's progress. We further explored a scheduling mechanism that trades off fairness for throughput, and described ways for implementing fairness-aware scheduling at different time scales and both in software and hardware.

Our experimental results demonstrate the significance of fairness-aware scheduling for heterogeneous multi-cores. We report substantial improvements in fairness over pinned scheduling (current multi-core schedulers) and throughput-optimized scheduling (current state-of-the-art in heterogeneous multi-core scheduling for system throughput), achieving average fairness levels of 86% for a 1B3S system running multi-program workloads. Fairness-aware scheduling also improves system throughput by 21.2% over pinned scheduling, while being within 3.6% compared to throughput-optimized scheduling. For homogeneous multi-threaded workloads, fairness-aware scheduling improves performance by 14% on average and up to 25%, and equal-progress and equal-time scheduling perform equally well. For heterogeneous multi-threaded workloads, equal-progress scheduling significantly outperforms equal-time scheduling, leading to an overall performance improvement of 32% on average over pinned scheduling.

Acknowledgements

We thank the anonymous reviewers for their constructive and insightful feedback. Kenzo Van Craeynest was supported through a doctoral fellowship by the Agency for Innovation by Science and Technology (IWT). Additional support is provided by the FWO project G.0179.10, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295. Experiments were run on computing infrastructure at the ExaScience Lab, Leuven, Belgium; the Intel HPC Lab, Swindon, UK; and the VSC Flemish Supercomputer Center.

References

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 298–309, June 2005.

[2] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *Journal of Instruction-Level Parallelism (JILP)*, 10:1–26, June 2008.

[3] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 290–301, June 2009.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct. 2008.

[5] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, Nov. 2011.

[6] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Design Automation Conference (DAC)*, pages 927–930, July 2009.

[7] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 511–522, June 2013.

[8] S. Eyerman and L. Eeckhout. System-level performance metrics for multi-program workloads. *IEEE Micro*, 28(3):42–53, May/June 2008.

[9] A. Fedorova, D. Vengerov, and D. Doucette. Operating system scheduling on heterogeneous core systems. In *Proceedings of the First Workshop on Operating System Support for Heterogeneous Multicore Architectures, held in conjunction with PACT*, Sept. 2007.

[10] R. Gabor, S. Weiss, and A. Mendelson. Fairness enforcement in switch on event multithreading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(3):34, Sept. 2007.

[11] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the Second Conference on Computing Frontiers (CF)*, pages 199–210, May 2005.

[12] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, Sept. 2011.

[13] J. Joao, M. Suleman, O. Mutlu, and Y. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–234, Mar. 2012.

[14] M. T. Jones. Inside the Linux scheduler: The latest version of this all-important kernel component improves scalability. http://www.ibm.com/developerworks/linux/library/l-scheduler/index.html, June 2006.

[15] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 125–138, Apr. 2010.

[16] 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 ACM/IEEE Annual International Symposium on Microarchitecture (MICRO)*, pages 81–92, Dec. 2003.

[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 *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 64–75, June 2004.

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

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

[20] 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 International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, Jan. 2010.

[21] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 317–328, Dec. 2012.

[22] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, May 2010.

[23] P. Michaud, A. Seznec, D. Fetis, Y. Sazeides, and T. Constantinou. A study of thread migration in temperature-constrained multicores. *ACM Transactions of Architecture and Code Optimization (TACO)*, 4(9), June 2007.

[24] NVidia. Variable SMP – a multi-core CPU architecture for low power and high performance. http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-Architecture-for-Low-Power-and-High-Performance-v1.1.pdf, 2011.

[25] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO)*, pages 81–93, Dec. 2004.

[26] G. Patsilaras, N. K. Choudhary, and J. Tuck. Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8, Jan. 2012.

[27] K. K. Rangan, M. D. Powell, G.-Y. Wei, and D. Brooks. Achieving uniform performance and maximizing throughput in the presence of heterogeneity. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 3–14, Feb. 2011.

[28] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 13–24, Feb. 2007.

[29] D. Shelepov, J. C. S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. HASS: A scheduler for heterogeneous multicore systems. *Operating Systems Review*, 43:66–75, Apr. 2009.

[30] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, Nov. 2000.

[31] S. Srinivasan, L. Zhao, R. Illikal, and R. Iyer. Efficient interaction between OS and architecture in heterogeneous platforms. *ACM SIGOPS Operating Systems Review*, 45:62–72, Jan. 2011.

[32] 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, Mar. 2009.

[33] 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 International Symposium on Computer Architecture (ISCA)*, pages 213–224, June 2012.