2015 IEEE 21st International Conference on Parallel and Distributed Systems

Contention-Aware Scheduling for Asymmetric Multicore Processors

Xiaokang Fan Yulei Sui Jingling Xue
Programming Languages and Compilers Group
School of Computer Science and Engineering, UNSW Australia

***Abstract*—Asymmetric multicore processors (AMPs) have been proposed as an energy-efficient alternative to symmetric multicore processors (SMPs). However, AMPs derive their performance from core specialization, which requires co-running applications to be scheduled to run on their most appropriate core types. Despite extensive research on AMP scheduling, developing an effective scheduling algorithm remains challenging. Contention for shared resources is a key performance-limiting factor, which often renders existing contention-free scheduling algorithms ineffective.**

We introduce a contention-aware scheduling algorithm for ARM's big.LITTLE, a commercial AMP platform. Our algorithm comprises an offline stage and an online stage. The offline stage builds a performance interference model for an application by training it with a set of co-running applications. Guided by this model, the online stage schedules a workload by assigning its applications to their most appropriate core types in order to minimize the performance degradation caused by contention for shared resources. Our model can accurately predict the performance degradation of an application when co-running with other applications with an average prediction error of 9.60%. Compared with the default scheduler provided for ARM's big.LITTLE and the speedup-factor-driven scheduler, our contention-aware scheduler can improve overall system performance by up to 28.32% and 28.51%, respectively.

***Keywords*—Asymmetric Multi-core Processor, performance interference, contention-aware scheduling, regression model.**

I. Introduction

As an alternative to homogeneous multicore processors, single-ISA heterogeneous multicore processors have been proposed to achieve higher performance with lower energy costs for applications with diverse architectural requirements. Single-ISA heterogeneous multicore processors, also known as asymmetric multicore processors (AMPs), typically consist of several high-performance *big* cores and a large number of energy-efficient *small* cores on a single chip. Big cores are designed with a sophisticated microarchitecture (e.g., with a high clock frequency and a complex out-of-order pipeline). In contrast, small cores are less complex (e.g., with a lower clock frequency and an in-order pipeline). Both core types share the same instruction set, so that a program can run in a consistent manner on any core.

Exploiting asymmetric features to achieve good performance-energy trade-offs for multicore processors has gained substantial interest in both academia and industry over the past few years. Recent research [12, 15, 16, 17, 18] has demonstrated potential benefits of AMPs over homogeneous multicore processors. Commercial industry solutions for single-ISA heterogeneous multicore processors include NVIDIA's Tegra 3 [23] and ARM's big.LITTLE [3]. Both designs provide different big and small core combinations to satisfy different performance and energy requirements of various applications.

The effectiveness of AMPs relies heavily on how to best schedule workloads according to their relative benefits derived from running on different core types. To improve the overall system throughput of applications running on AMPs, a few scheduling algorithms are proposed to map workloads to their most appropriate core types based on simulation [5, 16, 32], changing the frequency of some cores on symmetric multicore processors (SMPs) [27, 28] or modifying the internal workings of some cores [14].

The sampling based scheduling algorithms are initially proposed by Becchi et al. [5] and Kumar et al. [16] to decide which applications would use big cores more efficiently. Their scheduling algorithms are driven by the *speedup factor*, which is the improvement of an application's performance on a big core relative to a small core. These algorithms periodically sample the cycles per instruction (CPI) of an application on both core types to determine the relative benefit for the application to run on a big core.

HASS [28] represents an alternative approach to obtaining the speedup factor using offline profiling of an application to give a static hint for online scheduling. Its speedup factor is determined by estimating the last-level cache miss rate. This work was later extended by considering both efficiency and thread-level parallelism (TLP) for sequential and parallel applications [27]. In their experiments, dynamic voltage/frequency scaling (DVFS) technique is applied to change the frequency of some cores on SMPs. As a result, the cores in their setting differ only in terms of execution frequency.

Instead of sampling and offline profiling, the bias scheduler [14] performs dynamic scheduling by using core stalls to characterize the potential benefits of scheduling an application on a big core over a small core. Applications that exhibit frequent memory and other resource stalls are mapped to small cores, while applications whose CPI is dominated by execution cycles rather than stalls are mapped to big cores. To emulate an asymmetric system accurately, the authors have modified some cores in SMPs so that they are different only in the number of micro-ops retired per cycle.

Recently, the research on performance impact estimation (PIE) [32] shows that the previous approaches that map

1521-9097/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPADS.2015.98

memory-intensive workloads to small cores and compute-intensive workloads to big cores may result in suboptimal scheduling. Their dynamic scheduling mechanism collects profiling information when executing an application on any one core type to predict the performance on the other core type. This prediction allows appropriate scheduling adjustments to be made at runtime. The evaluation is conducted on a simulator where the proposed mechanism requires some profiling information, such as the inter-instruction dependency distance distribution, which cannot be collected on existing commercial cores and requires specialized hardware support [24].

A. Insights

Despite extensive research in AMP scheduling, how to design and implement an effective scheduling algorithm in a real asymmetric multicore system remains challenging. One of the fundamental problems facing the AMP scheduling designers is the fact that any single core, as part of an on-chip AMP system, is not an independent processor but sharing resources (e.g., cache, memory bus and memory controller) with other cores. The competition for shared resources causes performance interference of co-running applications, and this contention can significantly degrade their performance relative to what they could achieve in a contention-free environment.

Figure 1 shows four representative applications chosen from the 21 programs in Table III to highlight the performance degradations when co-running relative to solo runs (contention-free) on both big and small core types. The experiments are conducted on ARM's big.LITTLE, a commercial AMP system, which has two big cores and three small cores with different microarchitectures (Table I). For an application $\mathcal{A}$, 80 workloads are generated with each workload consisting of $\mathcal{A}$ and four randomly selected ones from the remaining 20 applications. The performance interference is observed by first binding $\mathcal{A}$ to a big core to co-run with the four applications randomly scheduled on the other four cores, and then measure the interference by binding $\mathcal{A}$ to a small core.

From the four representative applications, we can see that for some applications like `253.perlbmk`, the contention has relatively small impact on performance, resulting in small differences between the best and worst slowdowns. However, many applications such as `179.art` are observed to have a big difference gap between the best and the worst slowdowns on both core types. Applications like `176.gcc` have higher performance slowdowns due to contention when scheduled to a small core. In contrast, applications `255.vortex` have higher slowdowns when scheduled to a big core.

The performance interference among co-running applications can also affect the scheduling decisions made by previous contention-free approaches discussed earlier. As shown in Figure 2, when an application co-runs with other applications, its speedup on a big core over a small core may have a large deviation compared to the speedup measured in a contention-free environment. Thus, the existing speedup-factor-driven approaches that ignore the performance interference may lead to ineffective scheduling.

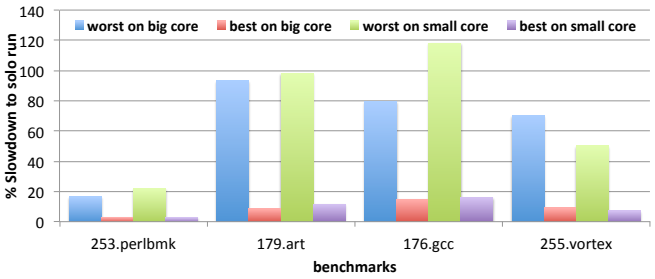


Fig. 1: The performance slowdowns relative to solo runs for different co-running workloads on big and small cores.

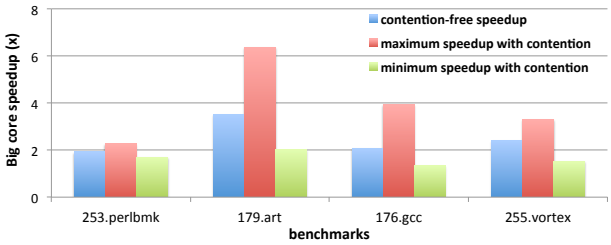


Fig. 2: Speedups of a big core over a small core under three scenarios: contention-free speedups, maximum speedups with contention, and minimum speedups with contention.

B. Challenges

Developing practical contention-aware scheduling algorithms in real asymmetric multicore systems is challenging. The microarchitectures of heterogeneous cores in a real AMP system differ in many ways, ranging from pipeline to memory hierarchy. These asymmetric features were not addressed inadequately in the past [5, 16, 27, 28, 32]. Moreover, although contention-aware scheduling in symmetric multicore processors has been extensively studied [11, 20, 33, 34, 35], the algorithms used in SMPs cannot be directly applied to AMPs. For example, DI [35], one of the representative SMP contention-aware scheduling algorithms, distributes the workloads such that the last level cache miss rate is evenly distributed among the caches. However, as different types of cores have different cache characteristics (Table I), mapping applications to cores with an evenly distributed last level cache miss rate will be ineffective in a real AMP system.

C. Our Solution

In this paper, we present a contention-aware scheduling approach to improving the overall system performance in a commercial asymmetric multicore architecture: ARM's big.LITTLE. The novelty of our approach lies in taking advantage of the performance interference relations to support effective dynamic scheduling by selecting an appropriate application-to-core mapping in a real asymmetric system. As shown in Figure 3, our scheduling strategy comprises (1) an offline stage to build a performance interference model that extracts the interference relations by using lightweight training

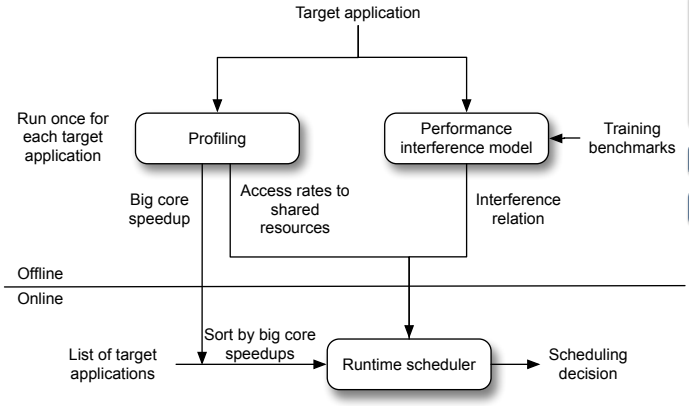


Fig. 3: A contention-aware scheduling framework for AMPs.

and (2) an online stage that schedules a set of applications to the most appropriate core types by considering both the speedup factor and the predicted performance interference.

For each application $\mathcal{A}$ in a candidate application list L, the offline stage builds a performance degradation function of $\mathcal{A}$ by applying regression analysis to determine the coefficients related to various shared resources (including cache, bus and memory controller) when co-running $\mathcal{A}$ with different training benchmarks. During the online stage, L is first sorted based on the big core speedup of each application in L. Then the dynamic scheduler maps an application $\mathcal{A}$ from L to an idle core by considering its interference impact when co-running $\mathcal{A}$ with the existing on-core applications.

The following are the key contributions in this paper:

- We present a contention-aware scheduling approach in a real commercial single-ISA heterogeneous multicore system to improve the overall system performance.
- We propose a new two-stage scheduling framework: an offline performance interference model that can accurately analyze the contention correlations among co-running applications and an efficient online scheduler by considering predicted performance interference impacts.
- We evaluate our scheduling approach on ARM's big.LITTLE. Our model can accurately predict an application's performance degradation due to contention with an average prediction error of less than 10%. Compared with the default scheduler provided by the development board used and the speedup-factor-driven scheduler, our scheduler can improve the system throughput by up to 28.32% and 28.51%, respectively.

The rest of the paper is structured as follows. First, we give a brief introduction to ARM's big.LITTLE architecture (Section II). Next, we present our contention-aware scheduling approach with its offline interference model (Section III-A) and online scheduler (Section III-B). Then, our experimental results are discussed (Section IV). Finally, we present the related work (Section V) and conclude (Section VI).

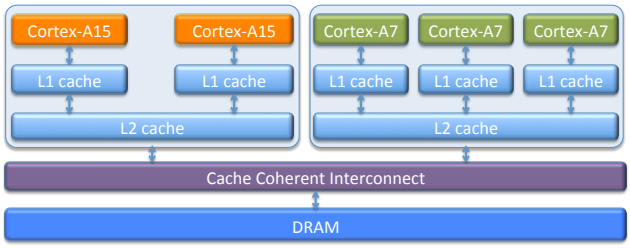


Fig. 4: ARM's big.LITTLE architecture.

Parameter	Cortex-A15	Cortex-A7
Pipeline	Out-of-order 15 – 24 stages	In-order 8 – 10 stages
Issue/fetch width	3/3	2/2
Branch predictor	2K-entry BTB/2-way	512-entry BTB/2-way
L1 I-TLB	32-entry fully associative	10-entry fully associative
L1 D-TLB	Two separate 32-entry fully associative TLBs for loads and stores, respectively	One single 10-entry fully associative TLB
L2 TLB	512-entry/4-way	256-entry/2-way
L1 I-cache	32KB/2-way/64B	32KB/2-way/32B
L1 D-cache	32KB/2-way/64B	32KB/2-way/64B
L2 cache	1MB/16-way/64B	512KB/8-way/64B

TABLE I: Microarchitectural differences between Cortex-A15 and Cortex-A7 cores in ARM's big.LITTLE.

II. ARM's big.LITTLE Architecture

Figure 4 illustrates the single-ISA heterogeneous architecture of ARM's big.LITTLE on which our scheduling framework is implemented. In big.LITTLE, cores of the same type are grouped together as a cluster. A single chip integrates a cluster of high performance Cortex-A15 cores [2] and a cluster of power-efficient Cortex-A7 cores [4] with modest performance.

Our evaluation is conducted on a big.LITTLE development board (V2P-CA15×2_CA7×2) which contains two Cortex-A15 cores and three Cortex-A7 cores. These two types of cores implement the same ARMv7-A instruction set. Every core has a private L1 instruction cache and a private L1 data cache. All cores in the same cluster share a L2 cache. Cache coherency between the two clusters is maintained via the CCI-400 cache coherent interconnect.

Table I lists the major microarchitectural differences between Cortex-A15 and Cortex-A7 cores. The A15 core is designed for high performance with a 3-way issue, out-of-order pipeline containing 15 – 24 stages, while the A7 core is designed for energy efficiency with a 2-way issue, in-order pipeline containing 8 – 10 stages. Furthermore, their branch predictors and cache configurations are distinctly different. The A15 core supports two separate 32-entry fully associative L1 Data TLBs for data loads and stores. In contrast, the A7

core has only one 10-entry fully associative L1 Data TLB for both data loads and stores. The unified L2 TLB of the A15 core is also more sophisticated than that of the A7 core. The two types of cores differ slightly in the L1 instruction cache and have the same L1 data cache. However, their L2 cache organizations differ greatly.

From Table I, we can see that a real AMP system has more complicated asymmetric features designed for different core types. The previous AMP scheduling algorithms that are developed based on a simulation [5, 16, 32] or by applying DVFS to the cores in SMPs [27, 28, 29] may not be sufficiently effective in accommodating these microarchitectural differences.

Through extensive experiments on a real AMP system, we find that contention is determined by multiple shared resources such as cache, bus and memory controller, all combined as a whole to create the performance degradation. Figure 8 shows the percent contribution that each of the factors has on the total degradation, with a detailed analysis given in Section IV.

III. Contention-aware Scheduling

In this section, we present our contention-aware framework for scheduling multiple applications in AMPs. As shown in Figure 3, every application is first trained by co-running it with a set of training benchmarks to determine the coefficients related to shared resources by using regression analysis. The online stage makes contention-aware scheduling to map the applications to the most appropriate core types by considering big core speedups and performance slowdowns predicted by the empirical interference model obtained offline.

A. Offline Interference Model

In the offline phase of each application $\mathcal{A}$, two key results are collected by answering two questions: (1) how aggressively does $\mathcal{A}$ access the shared resources in a contention-free environment and how does its performance degradation vary when $\mathcal{A}$ competes for resources with its co-runners?

The interference model first collects the individual pressure of $\mathcal{A}$ and its training benchmarks on each shared resources by binding each application to run on a certain type of core in isolation. The aggregate pressures on each shared resources are then obtained when $\mathcal{A}$ co-runs with another four selected training benchmarks.

Equation 1 denotes the aggregate pressure on a certain kind of shared resource R in a cluster $\mathcal{T}$ (either a big or a small core cluster):

$$P^{\mathcal{T}}(R) = \sum_{i=1}^{n} C_i^{\mathcal{T}}(R) \tag{1}$$

where n is the number of co-running applications in the cluster $\mathcal{T}$ and R represents one of the three critical on-chip resources: (1) shared cache among the cores in the same cluster, (2) shared bus that connects different clusters, and (3) shared memory controller among all the cores. The individual pressure on a shared resource R of the $i\text{-}th$ application in $\mathcal{T}$ is represented by $C_i^{\mathcal{T}}(R)$, which is the application's access rate (access count per second) to resource R collected by ARM's Streamline tool.

Applications running in the same cluster (*intra-cluster*) may have performance interference when competing for shared cache, bus, and memory controller, while the interference among the applications running in different clusters (*inter-cluster*) only comes from contention for the shared bus and memory controller. We differentiate the two cases by providing the following intra- and inter-cluster pressure models:

$$\textit{Intra-}P^{\mathcal{T}} = \alpha \cdot P^{\mathcal{T}}(cache) + \beta \cdot P^{\mathcal{T}}(bus) + \gamma \cdot P^{\mathcal{T}}(mem) + \sigma \tag{2}$$

$$\textit{Inter-}P^{\mathcal{T}} = \delta \cdot P^{\mathcal{T}}(bus) + \theta \cdot P^{\mathcal{T}}(mem) + \sigma' \tag{3}$$

where the coefficients α, β, γ, δ, θ, σ, σ' are to be instantiated using linear regression with training results.

The performance degradation function for an application $\mathcal{A}$ is defined by instantiating $\mathcal{T}$ in Equations 2 and 3 with big and small core types:

$$PD_{\mathcal{A}} = \begin{cases} \textit{Intra-}P^{big} + \textit{Inter-}P^{small} & \text{if } \mathcal{A} \text{ is on big core cluster} \\ \textit{Intra-}P^{small} + \textit{Inter-}P^{big} & \text{otherwise} \end{cases} \tag{4}$$

In order to select appropriate training benchmarks for building performance degradation functions that cover diverse contention interference cases, we use programs from CPU2006, MediaBench and MiBench as candidates. However, selecting a large number of training benchmarks without classifying their features can be ineffective or inaccurate.

Following [33, 34], we create a training set such that the benchmarks are evenly sampled based on their pressure on shared resources. A three-dimensional feature space is defined with each dimension representing a different shared resource (cache, bus or memory controller). Every candidate training benchmark is mapped into the feature space using its resource pressure. The feature space is evenly divided into $N_{cache} \times N_{bus} \times N_{mem}$ cubes, where N_{cache}, N_{bus} and N_{mem} are user defined. Candidates falling into the same cube are regarded as having a similar resource pressure. One point from each nonempty cube is sampled by adding it to the final training benchmark set. To reduce the training time, lightweight programs inputs (e.g., the train inputs for SPEC benchmarks) are used.

We use a SPEC benchmark, `175.vpr`, as an example to explain the key steps for building the performance degradation function for a specific application on ARM's big.LITTLE.

Step 1: Collecting pressure. Let $Progs$ be a set of programs including `175.vpr` and the benchmarks in the training set. For each program $i \in Progs$ on a core in cluster $\mathcal{T}$, we collect the individual pressures (access rates) of i on the three shared resources i.e., $C_i^{\mathcal{T}}(cache)$, $C_i^{\mathcal{T}}(bus)$, and $C_i^{\mathcal{T}}(mem)$. For each program, the average pressures are measured in ten solo runs per core type.

Step 2: Training. We generate 80 workloads, each containing four randomly selected benchmarks from the training set (Table III) to co-run with `175.vpr`. In the first 40 workloads, `175.vpr` is bound to a big core. In the remaining 40 workloads, `175.vpr` is bound to a small core. For the

Core type	Workload	Performance slowdown	Intra-pressure			Inter-pressure	
big	W_1	sd_1	$P_1^{big}(cache)$	$P_1^{big}(bus)$	$P_1^{big}(mem)$	$P_1^{small}(bus)$	$P_1^{small}(mem)$
	...	...		...		...	
	W_{40}	sd_{40}	$P_{40}^{big}(cache)$	$P_{40}^{big}(bus)$	$P_{40}^{big}(mem)$	$P_{40}^{small}(bus)$	$P_{40}^{small}(mem)$
small	W_{41}	sd_{41}	$P_{41}^{small}(cache)$	$P_{41}^{small}(bus)$	$P_{41}^{small}(mem)$	$P_{41}^{big}(bus)$	$P_{41}^{big}(mem)$
	...	...		...		...	
	W_{80}	sd_{80}	$P_{80}^{small}(cache)$	$P_{80}^{small}(bus)$	$P_{80}^{small}(mem)$	$P_{80}^{big}(bus)$	$P_{80}^{big}(mem)$

TABLE II: Performance slowdowns of `175.vpr` in 80 runs.

$i\text{-}th$ workload, the runtime performance slowdown sd_i of `175.vpr` and the corresponding aggregate pressures $P^{\mathcal{T}}(R)$ on shared resources of the workload are listed in the form of Table II on both big and small core types.

Let us take the first workload W_1 as an example. It consists of `175.vpr` and four training benchmarks: `433.milc`, `444.namd`, `456.hmmer` and `470.lbm`. In this workload, `175.vpr` and `470.lbm` are running on big cores while the other three are running on small cores. Under this co-running scenario, the performance slowdown (sd_1) of `175.vpr` is 167.28% on ARM's big.LITTLE with the following aggregate pressures on the three shared resources:

$$
\begin{aligned}
P_1^{big}(cache) &= C_{lbm}^{big}(cache) + C_{vpr}^{big}(cache) = 46.20\\
P_1^{big}(bus) &= C_{lbm}^{big}(bus) + C_{vpr}^{big}(bus) = 100.82\\
P_1^{big}(mem) &= C_{lbm}^{big}(mem) + C_{vpr}^{big}(mem) = 95.06\\
P_1^{small}(bus) &= C_{milc}^{small}(bus) + C_{namd}^{small}(bus) + C_{hmmer}^{small}(bus) = 49.79\\
P_1^{small}(mem) &= C_{milc}^{small}(mem) + C_{namd}^{small}(mem) + C_{hmmer}^{small}(mem) = 93.55
\end{aligned}
$$

Step 3: Building a degradation function. The data in Table II are then used during regression analysis to instantiate the coefficients in Equation 4. Finally, we obtain the performance degradation function for `175.vpr`:

$$
PD_{vpr} = \begin{cases}
0.1358 \cdot P^{big}(cache) + 1.5498 \cdot P^{big}(bus) \\
+0.0091 \cdot P^{big}(mem) + 0.4163 \cdot P^{small}(bus) \quad (\textbf{big cores}) \\
+0.0357 \cdot P^{small}(mem) - 14.4522 \\
- - - - - - - - - - - - - - - - - - - \\
0.4814 \cdot P^{small}(cache) + 0.9847 \cdot P^{small}(bus) \\
+0.0542 \cdot P^{small}(mem) + 0.6163 \cdot P^{big}(bus) \quad (\textbf{small cores}) \\
+0.0105 \cdot P^{big}(mem) - 56.2996
\end{cases}
$$

B. Online Scheduling

For a list of applications, L, the online scheduler aims to optimize the overall performance of all applications in L based on the offline information. Given the profiling information, an application $\mathcal{A} \in L$ with a relatively high (low) big core speedup under solo runs tends to be mapped to a big (small) core only if the predicted performance slowdown when co-running $\mathcal{A}$ with the currently on-core applications is under a predefined contention threshold.

Algorithm 1 shows the online stage algorithm for scheduling a list of applications in L, where the number of applications in L may be larger than the total number of cores available. For an idle core c, the applications in L are first sorted based on their big core speedups in descending order (lines 3 – 4) or ascending order (lines 5 – 6) according to the core type of c.

The sorted application list provides only a hint on which applications may use a big core more efficiently (than a small core). Then the performance interference is predicted for a set of applications W, which includes a chosen application $L[i]$ and the existing on-core ones (line 8) by using the performance degradation functions obtained during the offline stage (line 9). To predict the slowdown of an application $\mathcal{A}$ in W using $PD_{\mathcal{A}}$, we first calculate the aggregate pressure $P^{\mathcal{T}}(R)$ for each resource R on any cluster $\mathcal{T}$ using Equation 1. Then the collected aggregate pressures are substituted into the degradation function in Equation 4 to obtain the predicted slowdown $sd_{\mathcal{A}}$. Initially, when W contains fewer applications than the total number of cores, N, available, Equation 4 obtained earlier for N during the training phase is used.

Application $\mathcal{A}' = L[i]$ is selected (lines 12 – 14) if the computed performance slowdowns of the applications in W are all within a user-defined contention threshold whose performance impact is evaluated in Section IV-D. Otherwise, we re-evaluate the interference for the next candidate in L (line 7). If none of the applications in L satisfy the threshold condition, we choose the application $\mathcal{A}' = L[i]$ which causes the smallest total performance slowdown (recorded during each loop iteration) when it co-runs with the on-core applications (lines 16 – 18).

Finally, we map the selected application $\mathcal{A}'$ to the idle core c, remove $\mathcal{A}'$ from L, and mark c as busy (lines 19 – 22).

Let us use an example to explain our online scheduling algorithm. Suppose there are four applications: `255.vortex`, which is running on a big core, and `176.gcc`, `179.art` and `188.ammp`, which are running on the three small cores. We have a list L of other applications waiting to be scheduled to the last idle big core c. Let us go through Algorithm 1 to demonstrate how to map an application from L to run on c.

Suppose that after L has been sorted in line 4, `175.vpr` is at the head of L. The following steps are performed to determine whether `175.vpr` can be scheduled to c.

Step 1: Predict performance slowdowns. Following Equation 1, we first compute the aggregate pressures of `175.vpr` and the four on-core applications. The resulting aggregate pressures on three shared resources in both clusters are: $P^{big}(cache) = 23.58$, $P^{big}(bus) = 19.26$, $P^{big}(mem) = 11.27$, $P^{small}(cache) = 50.27$, $P^{small}(bus) = 73.91$, and

Algorithm 1: Online Scheduling

```
Procedure ONLINESCHEDULING()
begin
1   Let L be a list with m applications to be scheduled;
2   foreach c ∈ idleCores do
3       if c is a big core then
4           Sort the applications in L in descending
            order of their big core speedups
5       else
6           Sort the applications in L in ascending
            order of their big core speedups;
7       for i = 0; i < m; i++ do
8           Let W be a set including L[i] and the
            current running applications on all cores;
9           Let sd_A be the predicted slowdown of
            every application A in W computed using
            the performance degradation function PD_A;
10          Let pd_min be the minimum slowdown,
            initialized with the largest value possible;
11          Let S be a user-defined contention
            threshold;
12          if ∄A ∈ W : sd_A > S then
13              A' ← L[i];
14              break;
15          else
16              if pd_min > Σ_{j=0}^{|W|} sd_{W[j]} then
17                  pd_min ← Σ_{j=0}^{|W|} sd_{W[j]};
18                  A' ← L[i];
19      Map A' to core c;
20      W ← ∅;
21      Remove A' from L;
22      Remove c from idleCores;
```

$P^{small}(mem) = 34.06$. Then these aggregate pressures are substituted into the performance slowdown function of `175.vpr`, which is built during the offline stage (Section III-A). Finally, we obtain the predicted performance slowdown of `175.vpr` as 50.69% (line 9). Similarly, the predicted slowdowns, 2.83%, 35.16%, 30.95% and 36.68%, of the four on-core applications are calculated using their own degradation functions.

Step 2: Map an application to a core. Suppose that the user-defined contention threshold S is 60%. Then all predicted slowdowns are found to be within the threshold. As a result, `175.vpr` is selected as an candidate, $\mathcal{A}'$, to be scheduled to the idle core (lines 12 – 14). However, if S is decreased to 50%, the estimated performance slowdown of `175.vpr`, which is 50.69%, exceeds the threshold. As a result, `175.vpr` will not be selected as a candidate for scheduling at this stage. Instead, we continue to examine the other applications in L to evaluate their predicted slowdowns against S (line 7). If all predicted slowdowns of the applications in L exceed S when co-running with the current on-core applications, then we choose to schedule an application $\mathcal{A}'$ with the smallest total performance degradation pd_{min} when co-running $\mathcal{A}'$ with the existing on-core applications (lines 16 – 18).

IV. Evaluation

The objective of our evaluation is to demonstrate that our offline interference model has low prediction errors and our online stage is highly effective in scheduling a list of applications to improve overall performance in ARM's big.LITTLE. Our model can accurately predict the performance degradation with an average prediction error of 9.60%. Compared with the default scheduler provided by big.LITTLE and a speedup-factor-driven scheduler, our contention-aware scheduler can improve the overall system throughput by up to 28.32% and 28.51%, respectively.

A. Experimental Setup

We conduct our experiments on a recent ARM's big.LITTLE development board: *Versatile Express CoreTile* (V2P-CA15x2_CA7x3) which consists of two Cortex-A15 cores and three Cortex-A7 cores (Table I). All cores in the same cluster run at the same frequency. The frequencies of the Cortex-A15 cores and the Cortex-A7 cores are set to their default values, 1.2GHz and 1GHz, respectively. The development board runs Linux Kernel 3.10.33 [1].

We use the ARM Streamline tool integrated in ARM Development Studio to collect all the hardware performance counter values such as access rate to different shared resources. The Streamline tool interacts with a module called Gator that runs in the Linux kernel on the development board to collect the required data. The overhead caused by this kernel module is less than 1% of the execution time of each benchmark.

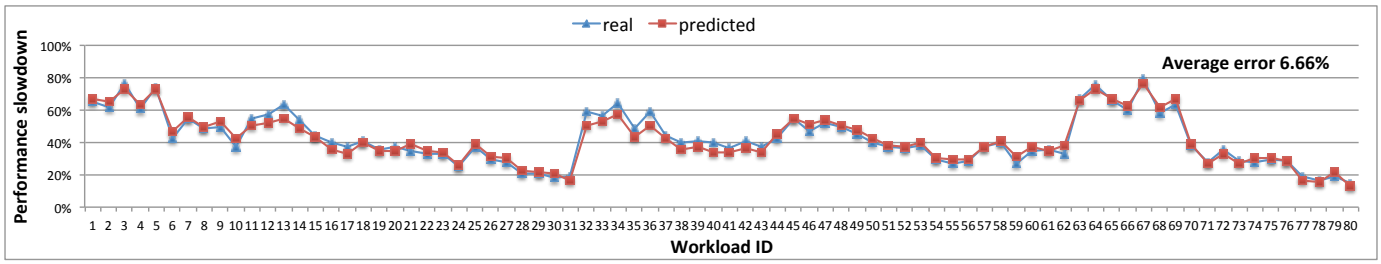
B. Benchmarks

Our training benchmarks are chosen from CPU2006, MediaBench and MiBench. The final training benchmark set is listed in Table III, which is selected using the sampling method mentioned in Section III-A. The three-dimensional feature space is evenly divided with $N_{cache} = 6$, $N_{bus} = 5$ and $N_{mem} = 5$. Finally, 28 training benchmarks are selected.

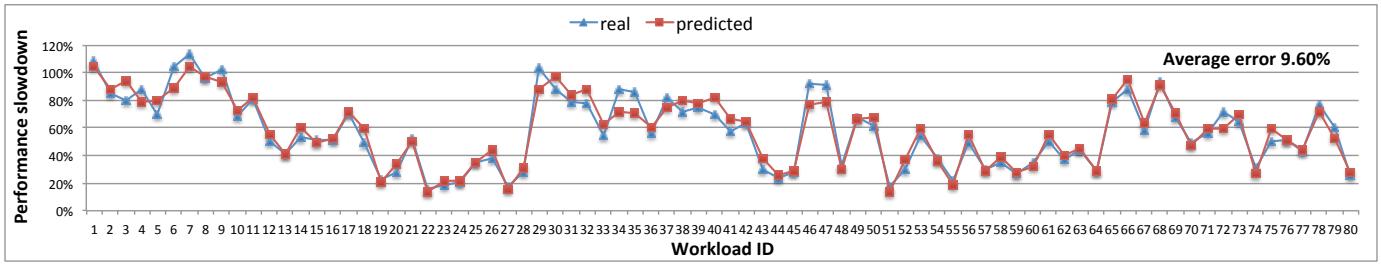
We use 21 programs from CPU2000 (Table III) for our target applications with the reference inputs used. All the program are cross compiled on the host machine (Linux kennel 3.13.0) under "GCC -O3" option, and then the binaries are deployed in the development board for conducting experiments.

C. Evaluating Interference Model

To evaluate the accuracy of the performance degradation function $PD_{\mathcal{A}}$ of an application $\mathcal{A}$ after training, we first measure the exact runtime performance slowdown of $\mathcal{A}$ when co-running $\mathcal{A}$ with other applications. Then, we compare



(a) Prediction accuracy on a big core with 80 workloads



(b) Prediction accuracy on a small core with 80 workloads

Fig. 5: Prediction accuracies of performance slowdowns with 160 randomly generated workloads.

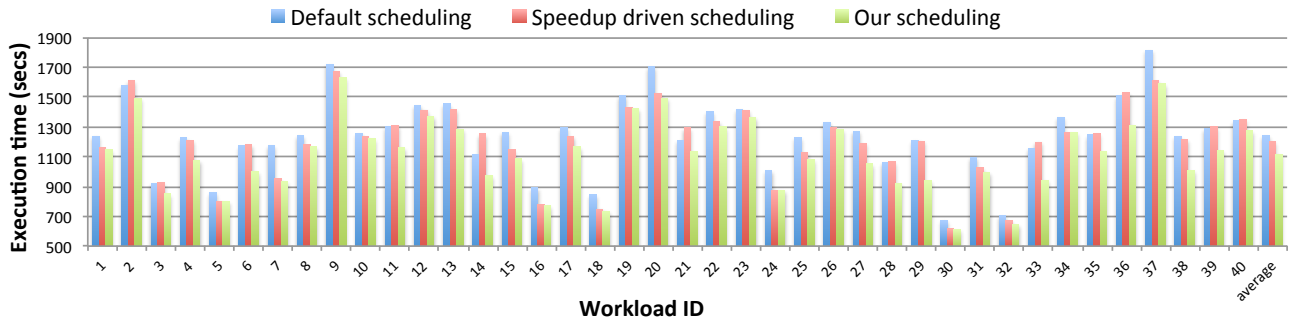


Fig. 6: Execution times of three schedulers with 40 randomly generated workloads with 10 applications per workload.

Training benchmarks	410.bwaves	416.gamess	433.milc	434.zeusmp
	435.gromacs	437.leslie3d	444.namd	445.gobmk
	447.dealII	456.hmmer	458.sjeng	464.h264ref
	470.lbm	482.sphinx3	basicmath	qsort
	tiffmedia	lame	jpeg	h264
	mpeg4	dijkstra	stringsearch	ispell
	rsynth	FFT	pgp	gsm
Target applications	164.gzip	168.wupwise	171.swim	172.mgrid
	173.applu	175.vpr	176.gcc	177.mesa
	179.art	181.mcf	186.crafty	188.ammp
	189.lucas	191.fma3d	197.parser	200.sixtrack
	252.eon	253.perlbmk	255.vortex	256.bzip2
	301.apsi			

TABLE III: Training benchmarks and target applications.

the exact runtime performance slowdown with the estimated slowdown produced by $PD_{\mathcal{A}}$ to validate the accuracy of the interference prediction model.

We generate 160 workloads with each of them made up of five randomly selected applications from the 21 applications in Table III. Figure 5 compares the accuracy of the estimated performance degradations (red lines) against the exact slowdowns during runtime (blue lines) on both big and small core types. For each of the first 80 workloads, one application running on a big core is randomly selected as the target application to evaluate the prediction error for the big core type (Figure 5(a)). Similarly, for the remaining 80 workloads, one application running on a small core is randomly selected from each workload to demonstrate the prediction accuracy for the small core type (Figure 5(b)).

In most cases, the estimated performance degradation is close to the real one measured during runtime. The prediction error for estimating the performance slowdown of an application on a big core ranges from 0.72% to 15.13%, with an average of 6.66%. The prediction error for a small core ranges from 0.40% to 19.45% with an average of 9.6%.

The eight applications chosen from the 21 applications

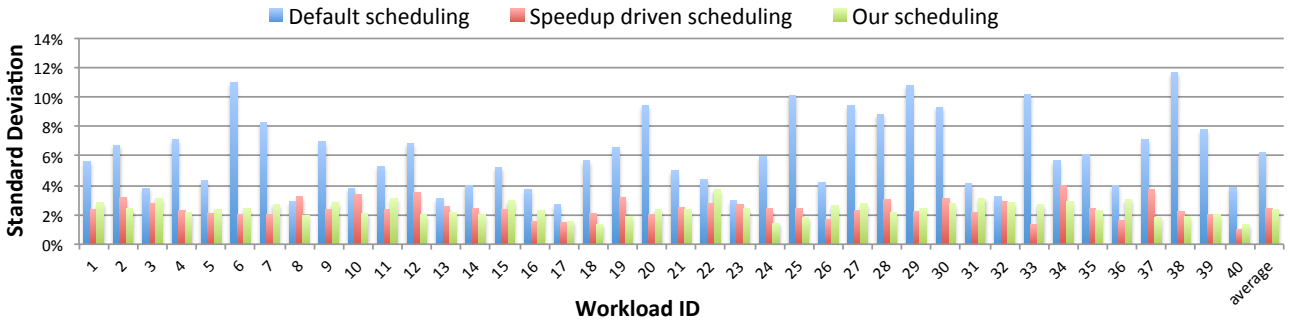


Fig. 7: Standard deviations of the same 40 workloads used in Figure 6 with each repeated for 10 runs.

shown in Figure 8 are the ones that have a maximum performance degradation over 50% on both big and small core types. Figure 8 shows the percent contribution that each of the factors has on the total degradation. The five factors, cache, bus_intra, mem_intra, bus_inter, mem_inter, correspond to the five coefficients $\alpha, \beta, \gamma, \delta$ and θ in Equations 2 and 3, which are instantiated by using regression analysis.

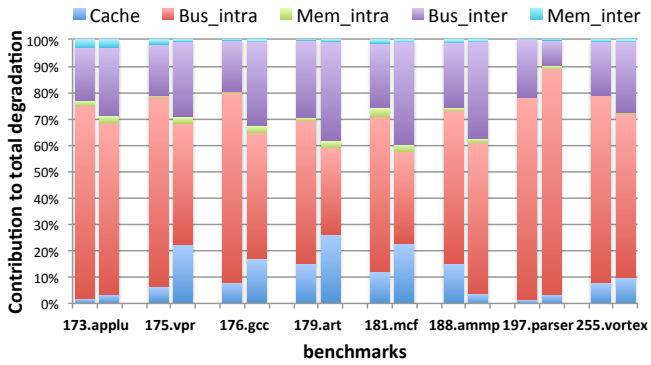


Fig. 8: Percent contribution from the five contention-impacting factors to the total performance degradations for eight selected applications.

D. Evaluating Scheduling

Our online scheduler is implemented by using the standard process affinity API in Linux to map a process to a certain core. For a list of candidate applications, our contention-aware algorithm is invoked to schedule an application to an idle core according to Algorithm 1.

The following two contention-unaware algorithms are used to compare with our contention-aware approach based on the total execution times for scheduling a list of applications.

- **Default scheduling** The default Global Task Scheduling (GTS) algorithm [13] provided by the development board with randomly ordered candidate applications.
- **Speedup-factor-driven scheduling** The algorithm that schedules an application with a higher big core speedup to a big core, and an application with a lower speedup to a small core.

We generate 40 workloads with each containing 10 randomly selected applications from the 21 applications in Table III. For a workload, the average execution time in 10 runs for each scheduling algorithm is recorded. The performance results and their corresponding standard deviations for the three schedulers are compared in Figures 6 and 7, respectively.

In most cases, our scheduler performs better than the other two. Compared with the default scheduler, the average speedup of our scheduler is 12.04% with a maximum speedup of 28.32%. Compared with the speedup-factor-driven scheduler, our scheduler can achieve an average speedup of 7.84% with a maximum speedup of 28.51%. Our contention-aware scheduler has smaller deviations (2.37% on average) in the 10 runs than the other two. The default scheduler has the largest deviation with an average of 6.22%, and the speedup-factor-driven scheduler has a deviation of 2.45% on average.

An important parameter that affects our contention-aware scheduler is the performance slowdown threshold as introduced in Section III-B, which is used to map an application to an idle core when the predicted slowdowns of the applications in a workload are within the user-defined value. Figure 9 compares the average execution times of the 40 workloads under a number of threshold values. The performance results vary across the threshold values selected. Best scheduling can be obtained by tuning an appropriate value, which is neither too small (leading to conservative scheduling due to its oversensitivity to the resource contention) nor too large (resulting in severe performance interference).

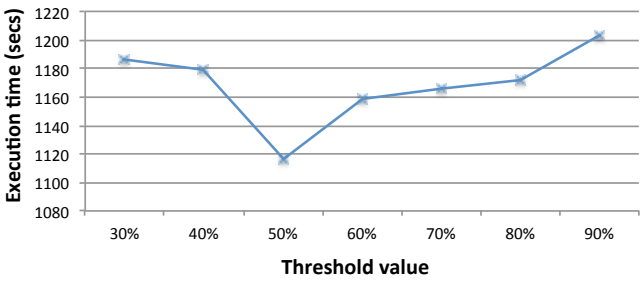


Fig. 9: Execution times under different contention threshold values (of S) in our content-aware scheduler (Algorithm 1).

V. Related Work

We limit our discussion to the most related work below:

a) Asymmetric Multicore Processors: A lot of work [5, 16, 24, 28, 31, 32] has been done on developing analytical performance models and contention-unaware scheduling algorithms for single-ISA heterogeneous multicore processors.

The speedup-factor-driven approaches proposed by Becchi et al. [5] and Kumar et al. [16] schedule an application to the most appropriate core type by periodically sampling the cycles per instruction of the application on different core types. However, the approaches scale poorly with the increasing core count. Instead of sampling, the offline approach [28] obtains the speedup factor using profiling information.

The phase-based tuning approach [29] uses control flow analysis to divide an application into code segments and group them into clusters such that all code segments in the same cluster show similar runtime characteristics. By sampling a few representative code segments from each cluster on each type of core, it obtains the runtime characteristic of each cluster and then makes scheduling decisions based on this information.

Performance impact estimation (PIE) [32] collects the profiling information of an application on one core type to estimate the performance of that application on the other core type. These authors demonstrate that memory-dominance scheduling [7, 14, 27] that maps applications with frequent memory-related stalls to small cores and compute-intensive applications to big cores may lead to suboptimal scheduling when memory intensity alone is not a good indicator. Later work by Pricopi et al. [24] improves PIE by building a predicting model that can predict both power and performance without the additional hardware support that PIE needs.

In [21], Moore et al. introduce an empirical strategy to automatically build estimation models that capture how a multithreaded program's performance scales with thread count and core type.

Van Craeynest et al. [31] have recently studied fairness-aware scheduling for single-ISA heterogeneous multicore processors. The approach accelerates threads on big cores based on two criteria, equal-time scheduling and equal-progress scheduling. Different from our goal to improve overall system performance, fairness-aware scheduling focuses on guaranteeing balanced progress of all threads. Our scheduling strategy can be complementary by improving the accuracy of their approach further with contention being considered.

b) Performance Interference: There is a large body of work to address the shared resource contention on symmetric multicore processors [8, 19, 25, 26, 30, 35, 36] and the contention of co-located applications in warehouse-scale computing [6, 11, 20, 33, 34].

As a representative case, cache contention has been extensively studied. Various cache partitioning algorithms are proposed to minimize the contention effects either via customized hardware solutions [25, 26] or software based page coloring approaches [8, 19, 30]. Some contention-aware scheduling algorithms [35, 36] are proposed for SMP systems where shared resources such as memory controller [22], on-chip interconnect [9], and prefetching [10] can significantly affect the overall system performance.

Several recent efforts on predicting, and scheduling warehouse applications are proposed to reduce the performance interference between co-located datacenter applications by considering shared resource contention. Bubble-Up [20] predicts the performance degradation between two applications for the shared memory subsystem. Bandit [11] focuses on bandwidth contention among several co-running applications that may hurt overall performance. Zhao et al. [33, 34] present a two-phase empirical model for predicting performance interference by considering both cache and bandwidth consumption using piecewise predictor functions.

VI. Conclusion

This paper presents a new contention-aware workload scheduler for asymmetric multicore processors. Our scheduling framework, evaluated in ARM's big.LITTLE, consists of an offline performance interference model for predicting the performance slowdown of an application when it co-runs with other applications, and an online stage for scheduling an application to the most appropriate core type based on predicted performance interference. Our design is simple and can be easily integrated in existing AMP systems. Compared with the default scheduler provided by big.LITTLE and speedup-factor-driven scheduler, our scheduler can improve overall system performance by up to 28.32% and 28.51%, respectively.

VII. Acknowledgement

The authors wish to thank the reviewers for their helpful comments. This work is supported by ARC grants, DP110104628 and DP130101970.

References

[1] Linaro stable kernel release for versatile express. https://releases.linaro.org/14.03/openembedded/vexpress-lsk.

[2] ARM. ARM Cortex-A15 MPCore processor technical reference manual, 2013. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438i/DDI0438I_cortex_a15_r4p0_trm.pdf.

[3] ARM. big.LITTLE technology: The future of mobile, 2013. http://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

[4] ARM. Cortex-a7 mpcore technical reference manual, 2013. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464f/DDI0464F_cortex_a7_mpcore_r0p5_trm.pdf.

[5] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06*, pages 29–40, 2006.

[6] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars. Enabling fair pricing on HPC systems with node sharing. In *SC '13*, pages 37:1–37:12, 2013.

[7] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *DAC '09*, pages 927–930. ACM, 2009.

[8] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *MICRO '06*, pages 455–468, 2006.

[9] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Application-aware prioritization mechanisms for on-chip networks. In *MICRO '09*, pages 280–291, 2009.

[10] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO '09*, pages 316–326. ACM, 2009.

[11] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *CGO '13*, pages 1–10, 2013.

[12] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *CF '05*, pages 199–210, 2005.

[13] B. Jeff. big.LITTLE technology moves towards fully heterogeneous global task scheduling. In *ARM White Paper*.

[14] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Eurosys '10*, pages 125–138, 2010.

[15] 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 *MICRO '03*, pages 81–92, 2003.

[16] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04*, pages 64–75, 2004.

[17] N. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *SC '09*, pages 1–12, 2009.

[18] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07*, pages 1–11, 2007.

[19] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA '08*, pages 367–378, 2008.

[20] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO '11*, pages 248–259, 2011.

[21] R. Moore, B. Childers, and J. Xue. Performance modeling of multithreaded programs for mobile asymmetric chip multiprocessors. In *ICESS '15*, 2015.

[22] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*, page 18, 2007.

[23] Nvidia. Variable SMP - a multi-core CPU architecture for low power and high performance, 2011. http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf.

[24] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin. Power-performance modeling on asymmetric multi-cores. In *CASES '13*, pages 1–10, 2013.

[25] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. *ISCA '06*, 34(2):167–178, 2006.

[26] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO '06*, pages 423–432, 2006.

[27] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys '10*, pages 139–152, 2010.

[28] 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(2):66–75, Apr. 2009.

[29] T. Sondag and H. Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *CGO '11*, pages 11–20, 2011.

[30] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: approximating L2 miss rate curves on commodity systems for online optimizations. In *ASPLOS '09*, volume 37, pages 121–132, 2009.

[31] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *PACT '13*, pages 177–187, 2013.

[32] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *ISCA '12*, pages 213–224, 2012.

[33] J. Zhao, H. Cui, J. Xue, and X. Feng. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2015.

[34] J. Zhao, X. Feng, H. Cui, Y. Yan, J. Xue, and W. Yang. An empirical model for predicting cross-core performance interference on multicore processors. In *PACT '13*, pages 201–212, 2013.

[35] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS '10*, pages 129–142, 2010.

[36] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 45(1):4, 2012.