Evaluating Asymmetric Multiprocessing for Mobile Applications

Songchun Fan Benjamin C. Lee
Duke University
{songchun.fan, benjamin.c.lee}@duke.edu

***Abstract*—Mobile workloads have diverse performance requirements. Some involve frequent user inputs and are computationally intensive while others are background services that minimally uses system resources. To exploit this diversity, architects propose asymmetric multiprocessing (AMP), which achieves both performance and energy efficiency by dynamically switching between high-performance and low-power cores. However, the benefits of existing AMPs that share main memory are limited by the high costs of switching between execution modes.**

Existing AMP architectures can exploit inter-application diversity but fail to exploit intra-application diversity. Exploiting the latter requires emerging AMP architectures that share the cache hierarchy and reduce switch latency by orders of magnitude. To explore the AMP design space, we propose a set of realistic mobile benchmarks on Android that consider user actions and inputs. We simulate the benchmarks and show that mobile apps benefit substantially from responsive AMPs that switch quickly to exploit fine-grained microphases.

I. Introduction

Mobile processors attempt to achieve two competing goals at the same time: high performance and low energy consumption. Traditional symmetric multiprocessing (SMP) attains the first goal, but fails the second one, as homogeneous cores are inefficient for the diverse performance requirements of mobile applications. Industry has taken two approaches to address SMP's limitations. The first approach, proposed by Qualcomm, provides independent power rails to each of the four cores, so that each core can adapt its voltage and frequency according to its workload. The second approach, taken by Samsung (ARM Big.LITTLE) and nVidia, uses additional low-power core(s) to handle less intensive tasks and powers off big cores.

With asymmetric multiprocessing (AMP), the mobile processor dynamically switches between big (out-of-order) and little (in-order) execution modes according to workload phases. Mobile app diversity provides AMPs many opportunities to switch. The first type of diversity is *inter-application diversity*. Some apps, such as video and web browsers, are compute-intensive and requires big cores; some apps, such as weather and calendar, require minimal computation and can be performed on little cores. Loosely coupled big and little cores that share memory are sufficiently responsive to inter-application diversity.

The second type of diversity exists inside individual apps, *intra-application diversity*. A mobile app normally contains more than one "activity window," allowing users to perform distinct types of actions such as scrolling, reading and typing. Different actions correspond to different patterns of computation. We observe that user input often triggers a short burst of processor activity, executing a few billion instructions in a few seconds. Input-triggered computation exhibits irregular control flow for which speculative and out-of-order execution perform poorly. For this type of computation, big cores perform no better, and sometimes worse, than little cores. Therefore, intra-app diversity provides opportunities for tightly coupled AMPs that share a cache hierarchy and switch quickly between execution modes.

Designing new microarchitectures for mobile AMP requires a comprehensive analysis of mobile workloads. Existing mobile benchmarks contain inter-app diversity but neglect intra-app diversity that arise from user actions and inputs. In this paper, we propose a set of mobile benchmarks that specifically include typical user activities and a systematic method to evaluate performance under different AMP designs. Specifically, we make three contributions:

Realistic Mobile Apps. We design and implement a set of realistic mobile benchmarks that are composed of Android apps. They include typical user actions, such as launching apps, scrolling down lists, reading contents and typing words. They also include foreground and background apps, such as games and music players. We integrate our benchmark apps with gem5 [7], a cycle-accurate simulator. To assist future work on mobile architecture design, we open-source the Android code and the simulator's disk images, checkpoints and configurations [2].

Responsive Asymmetric Multiprocessing. With our mobile benchmarks, we evaluate three AMP designs: 1) loosely coupled big and little cores with shared memory, 2) tightly integrated physical cores with a shared last-level cache, and 3) a single physical core with an adaptive datapath and shared cache hierarchy.

Efficient Big-Little Computation. We find that adaptive AMPs are responsive and use the little core for up to 47% of the instructions in mobile apps. Fine-grained switching produces 19-42% energy savings. The largest savings are seen for user actions, such as scrolling and reading, and for background services, such as a music service.

Collectively, we study realistic benchmarks that reflect intra-application diversity to show that emerging AMP architectures can save substantial energy without harming the performance of mobile applications.

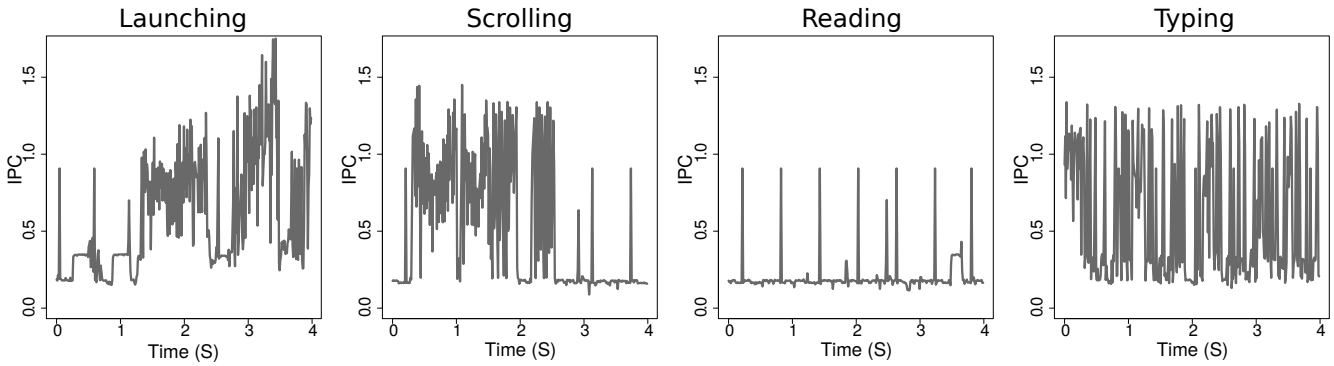

Fig. 1: Types of actions in one Twitter session

II. Mobile Benchmarking

Mobile apps are generally less computationally intensive; bursts of processor activity are often triggered by user inputs. Moreover, mobile apps often contain more than one activity window in order to provide multiple functionalities and let users perform distinct types of actions. Thus, an ideal benchmark set should capture not only the different performance requirements between apps, but also various user actions inside each app.

Existing mobile benchmarks neglect intra-app diversity, considering only end-to-end app execution. For example, BBench[13] provides a set of webpages, automatically loaded to test mobile web browsing behaviors. It also includes an interactive game which runs only on testbeds and requires manual inputs to execute, preventing the usage of cycle-accurate simulators.

MobileBench[29] includes photo viewing and video playback which are not interactive. Moby[15] provides popular mobile apps such as email, word processing, maps and social network. Typical operations in such apps, such as loading webpages and opening files, were simulated and do not include user inputs (click, scroll, type) that interact with apps. In contrast, we create a set of benchmarks for mobile apps that focus on typical user actions.

A. Application and User Actions

Consider the apps inside our phones. Some are simple (e.g., weather, photo viewer, or reader) and require only one type of user action, if any. Other apps, however, provide more functionality and allow users to perform multiple types of actions frequently. For example, users of a social app, such as Twitter, may frequently switch between multiple actions. These actions may include scrolling through a list of tweets, reading the details of a tweet, and typing a tweet. Reading does not require the mobile processor to do anything other than display content, but scrolling or launching new activities may trigger a sequence of computation. We expect different performance requirements for different actions.

To demonstrate this intuition, we conduct an empirical study. We use a hardware performance monitoring unit to record, in real time, the number of instructions committed per cycle (IPC) when a human user launches and uses Twitter (version 5.8.1 on Android JellyBean 4.3) on an ARM Versatile Express development board with one Cortex-A15 core activated [6]. During a 60-second session, the user performs several actions – launching new activity windows to open tabs or settings, scrolling to see more tweets, reading, and typing. We record user inputs and corresponding timestamps by reading the kernel input driver file and tracing click events.

Figure 1 presents selected user actions and the corresponding IPC time series. IPC shows distinct patterns for each action. Launching an activity window introduces a burst with a maximum IPC of 1.7. Scrolling produces a smaller burst with a maximum IPC of 1.5. Reading exhibits no computation except for the activity due to background sync tasks, which produce an average IPC of 0.2. Typing causes sustained IPC fluctuation. Differentiating such actions is important. If we were to view the Twitter session as a monolithic benchmark, end-to-end measurement would lose the rich information contained in various user actions.

In our setting, the mobile device is running the full system, with regular background tasks. For example, the IPC spikes when reading are caused by background network threads from the stock email client. This setting represents realistic mobile device usage and we will reproduce the same setting in our simulations.

B. Microbenchmarks

We create a set of microbenchmarks to represent typical user actions in a social app. In particular, the microbenchmarks correspond to the four actions in Figure 1.

- **Launching.** We create a set of activities and inject touch screen events that switch between them. A touch event is injected every 1 second to mimic a user launching different activity windows (e.g., clicking to view tweet details).
- **Scrolling.** We create a listview that automatically extends itself. Automatic swipes are injected to scroll

down the list. A swipe is injected every 500ms to mimic a user quickly scrolling down and browsing a list of content.
- **Reading.** We display an activity with text and pictures (e.g., reading a tweet). No specific inputs are injected.
- **Typing.** We create a textview and inject keyboard events to type (e.g., writing a tweet). Keyboard events are injected at a frequency of two letters per second.

To implement these benchmarks, we create "activities" within Android, loading activity text and pictures locally. We inject touches, swipes, and keyboard events using Android Instrumentation, an API that allows app developers to test their activity windows with emulated user behavior. This method requires access to application source code; our benchmarks are open-sourced.

Other methods use Android MonkeyRunner or write I/O events to the Linux input driver file. However, these methods require a time-stamp for each injected event and precisely specifying the time-stamp to trigger the right event during cycle-accurate, microarchitectural simulation is difficult. Alternatively, AutoGUI supports record-replay through VNC and may be useful once it becomes public [33].

C. Macrobenchmarks

In addition to microbenchmarks for input-triggered computation, we include two foreground tasks, a game and a mobile web browser. We benchmark a game in which the user controls a submarine by touching the screen. The game uses 2D graphics, which exercises the processor and neglects the GPU. We benchmark the web browser and use a script, BBench [13], that automatically loads local webpages.

Name	Scope	IPC(B)	IPC(L)	Inputs
Launching	MicroBenchmarks	1.01	0.80	Periodic
Scrolling		0.90	0.70	Periodic
Reading		0.84	0.70	None
Typing		1.05	0.76	Frequent
BBench	Web Browsing	0.98	0.76	Periodic
Music	Background Task	0.76	0.63	None
Submarine	Gaming	1.12	0.80	Periodic
SunSpider	Javascript	1.09	0.79	
Linpack	CPU	1.26	0.95	

TABLE I: Choices of Benchmarks

To compare mobile workloads against compute-intensive ones, we further deploy 0xbench [1] for testing smartphone performance. This open-source benchmark suite includes a Java implementation of Linpack and a Javascript benchmark called SunSpider.

Table I lists our benchmark and application suite. We classify benchmarks by the frequency of user inputs. We deploy the benchmarks on the gem5 simulator – see Section IV-B for detail. We provide deployment checkpoints, disk images, and methods in our open-source project [2].

III. Asymmetric Mobile Processors

AMP organizations can be classified into three categories: shared memory, shared last-level cache, and shared first-level cache. We abstract them and produce evaluation models by quantifying the costs of switching between execution modes.

A. Shared Memory

AMPs produce a performance and power gap between the big and little cores. The Tegra fabricates Cortex-A9 cores with different technologies such that big cores use faster transistors, which dissipate more static power, and little cores use slower transistors, which dissipate less static power [3], [4]. In contrast, the Exynos uses two microarchitectures, implementing big cores with Cortex-A15s and small ones with Cortex-A7s [8].

Both systems can be viewed as one cluster of big cores and one of little cores, each with a private last-level cache (LLC). Either the big or little cluster is active at any given time. Clusters have separate power domains, allowing big cores and their caches to power down when small cores are active, and vice versa.

We abstract this heterogeneous architecture with one out-of-order and one in-order core, each using private LLCs but sharing DRAM. A big/little switch transfers a thread from one core to another. Because the clusters' LLCs are private, the dirty cache lines of the currently active LLC are flushed to DRAM before switching to ensure coherence. This flush dominates the switching latency.

Switching Cost – 10K cycles. The switching delay is computed from the size of dirty LLC lines and the memory bandwidth. If LLC capacity is 512KB, memory bandwidth is 12.8GB/s, and 25% of cache lines are dirty,[1] a 1GHz core must wait 10K cycles (= 25% $\times$ 1GHz $\times$ 0.5MB /12.8GBps) for the switch to complete.

Although we optimistically assume transfers at peak memory bandwidth, 10K cycles is a useful, order-of-magnitude estimate of switching latency for asymmetric cores that share main memory. With this latency, cores are responsive to coarse-grained system phases (e.g., phone's standby/active modes), but further responsiveness is constrained by switching overheads.

B. Shared Last-Level Cache

Although current AMPs share memory, we envision next-generation AMPs with a shared last-level cache. Compared with shared memory, shared LLCs improve performance in several ways. First, the active core benefits from more cache lines since the aggregate size of the shared LLC is larger than that of a statically partitioned one. For example, the ARM Big/Little allots 1MB and 0.5MB of L2 for its big and little cores, whereas a shared cache would provide 1.5MB to

[1]This number is measured by our mobile workloads, on average.

be used by active cores [6]. Second, a shared LLC reduces switching latency because flushing smaller private caches into the LLC is much faster than flushing dirty lines of a large LLC into memory.

Switching Cost – 500 cycles. If 25% of lines in a 32KB L1 cache are dirty, flushing the L1 into the L2 requires 500 cycles (= 25% $\times$ 32KB / 16B/cy). By avoiding writes to main memory, AMPs with the shared LLC reduces switching latency by two orders of magnitude. Such an AMP can respond to coarse-grained application phases with distinct resource demands [20].

C. Shared First-Level Cache

Recently proposed microarchitectures adapt the datapath between out-of-order (OOO) and in-order (IO) execution, providing big and little modes for a single physical core. Caches and their contents remain useful and valid across datapath transitions. In effect, big and little cores share the cache hierarchy, beginning with the L1. Core Fusion combines multiple cores into a more capable one [16], [18]. MorphCore reconfigures an OOO core into a highly multithreaded IO core [17]. Composite Cores use a shared front-end that feeds instructions into one of two backends, one IO and the other OOO [26].

Switching Cost – 30 cycles. Depending on the microarchitecture, switching latency can be negligible. Datapath transitions affect only architected state in registers, not the cache hierarchy. In one implementation, a switch flushes the pipeline, powers off OOO structures (e.g., physical register file), and resumes instruction fetch in IO mode. Switching latency is proportional to instruction window size. If the OOO datapath commits two instructions per cycle, flushing a pipeline with a half-occupied, 192-entry reorder buffer requires approximately 50 cycles.

In a second implementation, the datapath explicitly spills and fills architected state to and from the L1 cache, which requires 30 cycles in MorphCore. When switching between execution modes requires only tens of cycles, AMPs can respond to an application's fine-grained microphases [26].

IV. Methodology

We simulate, with gem5, varied AMP architectures to understand the impact of intra-app diversity on energy efficiency. We assume that only one core, either big or little, is active at any given time. We quantify the maximum percentage of instructions that can be executed on the little core that safeguards performance (i.e., instructions per cycle) yet improves energy efficiency. We consider an oracle that collects processor activity during app execution and, offline, evaluates big-little transitions with varying specifications and costs, as well as with perfect knowledge of the future.

First, we divide time into intervals. At the beginning of every interval, the oracle determines whether the other core would offer better performance or efficiency. For example, if the big core is currently active, the oracle switches to the little core in the next interval, if IPC(little) is similar to IPC(big). Three parameters affect the calculation of IPC.

Switching interval, measured in instructions, determines how frequently the oracle makes a switching decision. For example, suppose the interval is 1000 instructions and the oracle knows that, for the next 1000 instructions, IPC(little) = IPC(big) = 1. The little core executes instructions more efficiently with no performance penalty and the oracle considers a switch. Yet this switching decision is not final because, even if the little core seems capable, switching requires data migration and an additional delay that may cause performance to be lower than expected.

Switching cost, measured in cycles, quantifies the additional delays. Following the previous example, suppose that a big-little switch requires 500 cycles. To complete the next 1000 instructions, the little core requires 1500 cycles after accounting for switching cost whereas the big core requires only 1000. We define

$$\text{IPC}_{\text{new}} = \frac{\text{[Interval Insns]}}{\text{[Interval Insns]}/\text{IPC}_{\text{old}} + \text{Cost}},$$

which is the IPC after accounting for switching cost. In the previous example, performance clearly suffers since $\text{IPC}_{\text{new}} = 0.7$ and $\text{IPC}_{\text{old}} = 1.0$. Therefore, the oracle might not consider a switch. However, this is still not the final decision because a switching scheme might trade performance losses for efficiency gains.

Performance penalty specifies the performance loss that can be tolerated after switching. The oracle switches from big to little as long as IPC(little) is greater than or equal to penalized performance IPC(big)/Penalty. In our recurring example, the oracle would switch only if the penalty tolerance is at least 1.5$\times$. Altogether, the switching interval, switching cost, and performance penalty define a space of control parameters for switching between big and little cores.

A. Oracular Switching

With the parameters defined, the oracle identifies all switch points within the app with the following procedure. It first compares big and little IPC for each instruction interval to find those that satisfy conditions for a switch with respect to the penalty tolerance. For the original IPC time series in Figure 2(1), the oracle marks the time serials with "big" and "little" flags which suggest the most beneficial executing mode for each interval. If adjacent intervals are marked differently, they define an initial switch point. See points *A* and *B* in Figure 2(2).

The oracle then examine each big-to-little switch point based on the switching cost. Consider switch point *A*. Suppose that immediately after the switch, interval i_k reports IPC(big) = 1.0 and IPC(little) = 0.8. After applying a 500-cycle switching cost, IPC(little) = 0.57, which exceeds the

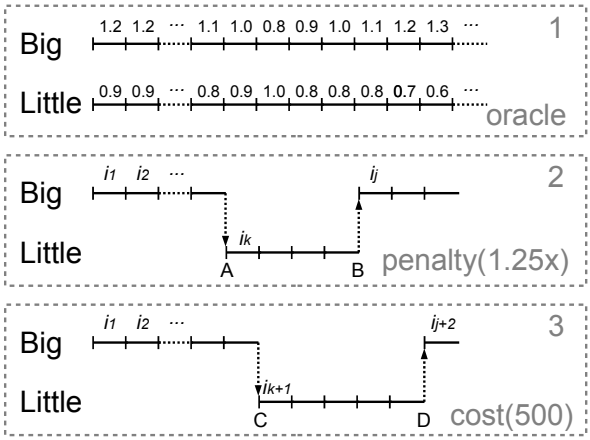

Fig. 2: Switching based on oracle knowledge

tolerated performance penalty such that the little core is no longer good for interval i_k. The oracle revises its decision, marks interval i_k big, and moves on to consider whether i_{k+1} should use the little core.

Similarly, the oracle examines little-to-big switch points. Consider point *B*. After applying a 500-cycle switching cost to interval i_j, IPC(big) $<$ IPC(little) $\times$ 1.25 and the big core is no longer suitable for interval i_j. The oracle revises its decision, marks interval i_j little, and moves on to subsequent intervals. After applying switching costs, the oracle obtains final switch points that differ from the initial ones – see points *C* and *D* in Figure 2(3). The final marks and switch points dictate total performance loss and energy savings.

The final analysis marks more intervals as little because switching to big is costly; the big core's performance advantage must be large enough to offset switching delays. Often, an initial little-to-big switch is discarded in the final analysis and the little core is used instead. Even though the little core's performance is low, the big core's performance is even lower after accounting for switching costs (e.g., interval i_{j+1} in Figure 2(3)). As costs increase, an AMP might use the little core more often because once a little core is used, high costs make a transition back to the big core hard to justify.

B. Simulation

Each mobile benchmark is installed in an Android (version 4.0.2) system image that is checkpointed immediately before the launching of a benchmark app. To study opportunities for fine-grained transitions, we collect performance statistics for intervals of 1000 instructions. The oracle uses IPC reported from big and little executions to determine the optimal execution mode for each interval.

Because the hardware register counters on the ARM development board cannot support nanosecond-level measurements, we use gem5 [7], a cycle-accurate simulator for evaluation. Gem5 supports full-system Android simulation for our mobile benchmarks. In Table II, we specify out-of-order and in-order cores for big-little asymmetry.

	Freq.	Width	#ALU	ROB	PRF	L1 I/D	L2 Cache
Big	1GHz	3	2	192	256	32KB	512KB
Small	1GHz	3	1	32	96	32KB	512KB

TABLE II: Big/Little Specs

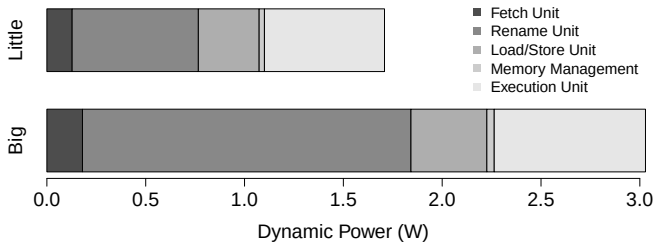

Fig. 3: Power Breakdown

We model an in-order core with issue logic that enforces first-in, first-out order. Without dynamic instruction scheduling, the datapath has fewer instructions in flight. Our datapath shrinks the physical register file and the reorder buffer to reflect less demand for these structures, but a more efficient in-order design would bypass them entirely. Our big and little configurations capture the out-of-order performance advantage and the in-order efficiency advantage.

Power. We use McPAT [24] to estimate dynamic power for big and little cores. Figure 3 shows the power breakdown for big and little cores when running BBench. Our in-order core saves much of its power in the rename because it reduces the size of the physical register file and the reorder buffer. However, because the rename was not completely removed, a small amount of power continues to be dissipated by this unit. Thus, power and energy savings that we report are conservative; an in-order core designed from first principles forgoes a rename unit and would dissipate even less power.

We compare power numbers from McPAT against power measurements collected, using on-board power meters, from an asymmetric ARM processor comprised of Cortex A15's and A7's [6]. When one big core is active and running BBench, voltage varies from 0.9 to 1.05V and average power is 1.10W. When one little core is active, voltage is 0.9V and average power is 0.36W. The big-little power ratio is approximately 2.5$\times$.

McPAT reports a lower ratio of 1.7$\times$. The simulated little core differs from the ARM Cortex A7 in two regards – rename and superscalar width. If the simulated little core were to bypass rename, McPAT would report a big-little power ratio of 2.6$\times$, which is more consistent with our ARM measurements. Without rename, our little core dissipates more power due to its wider datapath; it issues up to three instructions per cycle whereas the A7 issues only one. We evaluate sensitivity to superscalar width in later sections.

Finally, static power plays a large role in asymmetric processor efficiency. When the big core is active, the little core is idle and dissipates static power (and vice versa). The processor cannot use C-states to eliminate static power since that would require separate power supplies and millisec-

onds for wake-up. Instead, an asymmetric processor reduces dynamic power with idleness and reduces static power by power-gating with low-leakage PMOS headers, which wake in a few cycles.

We estimate static power as 30% of dynamic power or, equivalently, as 23% of the total. This estimate is consistent with recent processor implementations: IBM Power 7 static power is 25% of the total in a 45nm technology [37]. Circuit designers and CAD optimizers tune supply and threshold voltages to balance static and dynamic power. If static power were too large a fraction of the total, designers would increase Vth at the expense of circuit speed [14], [27].

V. Evaluation

We evaluate three asymmetric processor organizations: shared memory, shared last-level cache, and shared first-level cache with an adaptive datapath. By abstracting these designs into switching costs, we compare their energy savings for our suite of mobile benchmarks. We also assess sensitivity to other parameters such as interval length, IPC prediction accuracy, superscalar width, and voltage/frequency.

Parameters. Initially, we set the interval length to be 1000 instructions, which illustrates microphase behavior [26] and exercises the adaptive processor with a shared first-level cache. Later, we explore little core utilization and its sensitivity to this parameter.

Switching costs are dictated by data movement costs in each asymmetric processor design strategy: (a) 10K cycles for shared memory, (b) 500 cycles for shared last-level cache, and (c) 30 cycles for shared first-level cache. Although switching cost normally includes delays from data movement and wake-up, we assume data movement delay hides wake-up delay as is the case when power-gating.

We consider a range of tolerable performance penalties, measured in terms of the performance ratio between big and little cores. If the performance penalty were to exceed this ratio, little core utilization would reach 100%. Our evaluation sets the penalty below this ratio, ranging from 1.00 to 1.45.

Questions and Metrics. Our evaluation answers three questions about asymmetric multiprocessors for mobile workloads. First, is it necessary to have a little core? To quantify little core utilization, we use the method described in the previous section to find all the intervals that are marked little and compute the percentage. Second, is the little core too slow? We calculate total slowdown to quantify the end-to-end app delay when permitting little core execution. Third, what can we gain from the little core? We calculate total energy savings, which measures the ratio of energy consumed by big-little cores to the ratio of energy consumed by a big core alone.

A. Case Study with Scrolling

Scrolling is a fundamental microbenchmark for mobile phones. It is characterized by periodic user inputs that trigger

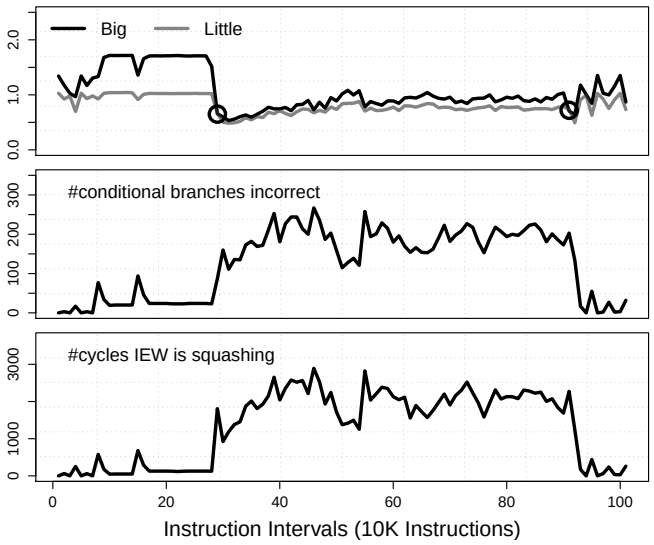

Fig. 4: Impact of branch misprediction

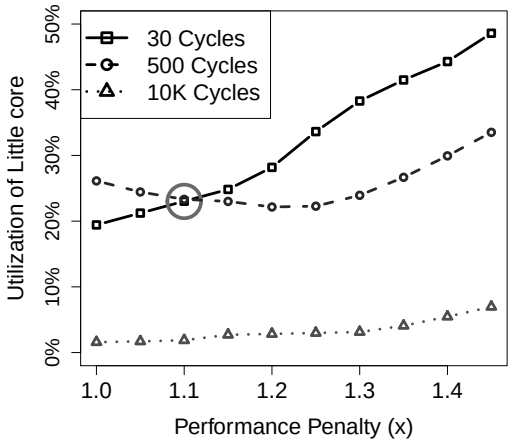

Fig. 5: Utilization of the little core for scrolling

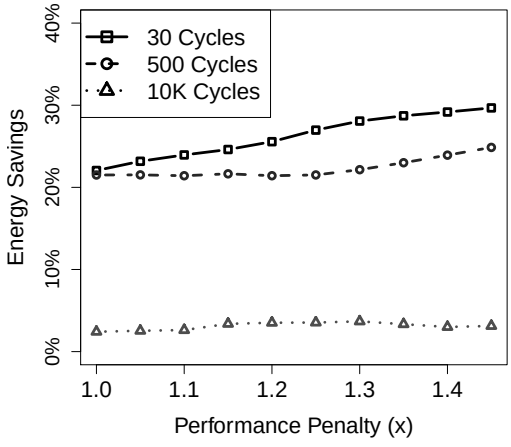

Fig. 6: Total energy saved, relative to big core energy

bursts of computation. Once the user touches the screen, the processor is active for only two to three seconds to execute a few billion instructions. In many instruction intervals, the little core performs as well as, if not better than, the big core.

For example, Figure 4 illustrates the impact of branch mis-prediction. Bursty computation when scrolling means that branch predictor training is less effective and speculative execution is often wasted in the big core. Such short and bursty computation, also observed in game and web browser benchmarks, has a direct impact on how an asymmetric multiprocessor is exploited.

Lower switching costs increase little core utilization, as illustrated by Figure 5. The adaptive core (30-cy) and shared-LLC (500-cy) strategies use the little core for 20-25% of instruction intervals, even under stringent constraints on performance penalties. For short intervals, today's shared-memory (10K-cy) strategy is ineffective and utilizes the little core for only 5% of intervals.

The cross-over point between 30-cy and 500-cy strategies highlights a counter-intuitive observation. Sometimes, the 500-cy strategy uses the little core more often. This is because switching back to the big core is difficult when switching costs are high and performance penalties cannot be tolerated – see Section IV-A.

Figure 6 shows energy savings of up to 30% from responsive AMPs. Recall that our power analysis for the little core is conservative, which makes the reported energy savings conservative as well. Big and little energy is calculated by multiplying core power and the number of active cycles in each mode. In addition, switching energy is calculated by multiplying big core power and the number of cycles spent switching.

For short intervals, there is no case in which today's AMPs are effective. The 10K-cy strategy performs worst for all three metrics, showing low utilization and low energy saving. Containing only 1,000 instructions, each interval is too short when compared to the switch cost and opportunities to switch are limited.

B. Generalizations with Benchmark Suite

For scrolling, the little core can be well utilized and save energy without harming performance. To generalize this conclusion, we present results for little core utilization and energy savings across all the benchmark apps. To simplify the illustration, we fix the performance penalty to be 1.15x. In practice, the value of the performance penalty can be tuned to achieve certain requirements (*e.g.*, an energy budget).

Figure 7 shows utilization of the little core. Mobile apps are more little-core friendly than computationally intensive benchmarks such as Javascript and Linpack. Apps that do not process user input, such as Read and Music, require little processor activity and utilize the little core most. Other apps receive periodic user input that demands computation, such as Launch and Scrolling, but still use the little core 15-24%

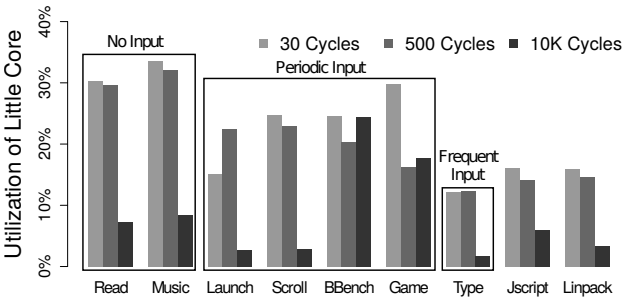

Fig. 7: Little core utilization across apps

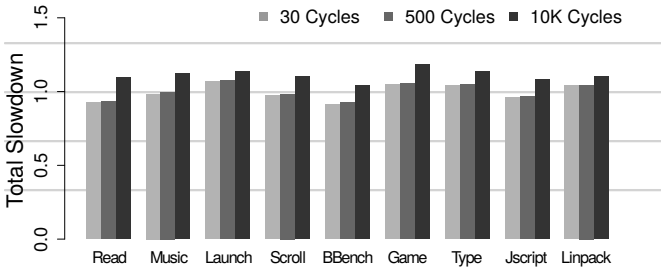

Fig. 8: Total slowdown across apps

of the time. Typing is a notable exception and uses the big core to process frequent user inputs. The 30- and 500-cy strategies use the little core often (24% and 22%) while the 10K-cy strategy uses it less (9%).

Figure 8 shows total slowdown when the tolerable performance penalty is set to 1.15x. Fast, fine-grained switches between execution modes can produce negative slowdowns (*i.e.*, speedups) because the little core can out-perform the big one (*e.g.*, due to branch mis-prediction penalties). Slow, coarse-grained switches may harm performance but slowdowns are capped by the penalty tolerance of 1.15x.

Figure 9 shows energy savings. Both the 30- and 500-cy strategies reduce energy by as much as 38%. Reading and Music are much less computationally intensive than other tasks, providing the most opportunities for the little core. The three strategies offer energy savings of 24%, 22% and 6%, on average, when the penalty tolerance is stringent. When performance targets are relaxed and permit up to a 1.45x penalty, the 30- and 500-cy strategies offer energy savings of 31% and 25% (average), and 42% and 39% (best case).

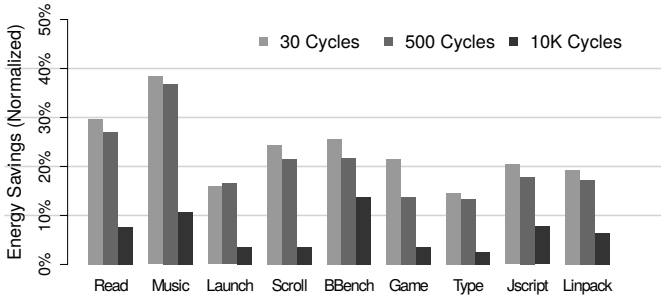

Fig. 9: Energy savings across apps

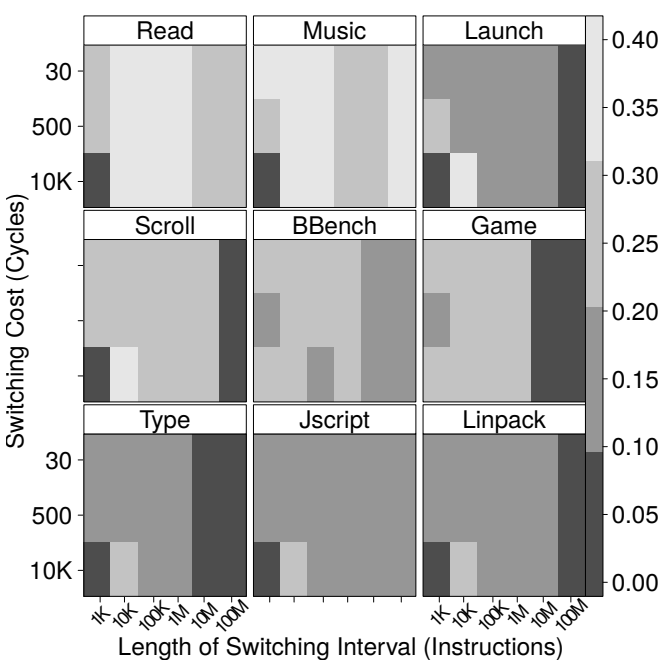

Fig. 10: Impact of switching cost, interval length on energy savings

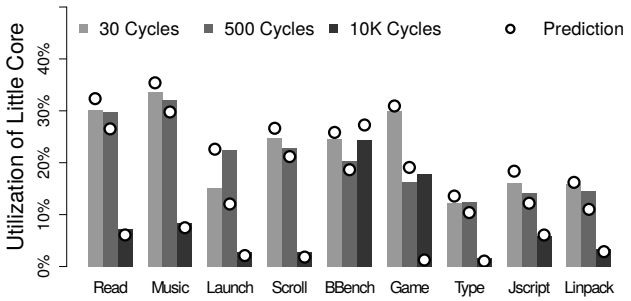

Fig. 11: Impact of prediction errors on little core utilization

C. Sensitivity to Management Parameters

Interval Length. The interval length determines the switching granularity. Figure 10 shows the energy savings for varied interval lengths. The brighter the color, the greater the energy savings. The 30- and 500-cy strategies enable the shortest intervals and provide the greatest savings; the top-left corner is brighter. The longest intervals do not reduce energy costs; the right-most columns are darker. The 30- and 500-cy strategies are insensitive to interval length, but the 10K-cy strategy saves energy only when interval length is greater than 10K instructions.

Apps that process periodic user input, such as Launching, Scrolling, and Submarine, are sensitive to switching interval and cost. As the interval length increases, the performance ratio between big and little cores in each interval approaches the average ratio across all intervals (1.0 to 0.7) and reduces switching opportunities. When the interval includes 10M or 100M instructions, the little core is rarely used and energy savings are low as indicated by the dark columns on the right. Finally, compute-intensive apps, such as Linpack, Javascript and Typing, are insensitive to interval and cost – they rarely use the little core and save little energy.

Performance Prediction Error. Thus far, our oracle-based evaluations have demonstrated little core utilization and its energy efficiency. In reality, perfectly knowing processor performance ahead of execution is impossible. It is possible, however, to build a model that predicts the performance of the big/little core with historical statistics obtained from hardware performance counters. The predicted IPCs can then be used to determine if a switch should happen for the next instruction interval. Such prediction models introduce error, and in this subsection, we discuss the impact of misprediction to the little core utilization.

To model the relationship between inputs from various hardware performance counters, linear regression models are often used. In such models, error terms are independent and identically distributed, following a normal distribution with mean of zero and variance of σ^2, To capture this effect in our oracle, we add Gaussian white noise (mean = 0, sd = 0.06) to the oracle's IPCs, mimicking predicted IPCs with errors with a distribution that is carefully tuned to correspond with related work [26]. We repeat this process ten times and, each time, recalculate little core utilization.

Figure 11 shows the impact of prediction error on little core utilization. Each bar shows oracle-based utilization while each circle shows prediction-based utilization. Most of the time, the impact of misprediction is negligible. An outlier exists in the 10K-cycle strategy running Submarine Game. Utilization with predicted IPCs tend to be much lower than that of the oracle-based result.

We explain the outlier as follows. The 10K-cy strategy with 1K switching interval normally leads to modest utilization of the little core. Utilization that is larger than expected (e.g., nearly 20% for Game) is explained by rare cases in which the oracle switches from big to little and then stays in little mode for long periods due to high switching overheads. Prediction errors avoid becoming stuck on the little core by removing the rare big-to-little switches.

Other than this outlier, the predicted little core utilizations for 30- and 500-cycle strategies deviate from the oracle by a maximum of 10.4%. This means that our oracle-based evaluation represents a realistic estimation of little core utilization. Based on existing prediction methods, the energy savings in our results are achievable.

D. Sensitivity to Design Parameters

Superscalar Width. By reducing the width of the little core, the performance gap between big and little will grow and little core's utilization will decrease. On the other hand, energy savings might increase because the little core now dissipates much less power. Figure 12 shows utilization of a 1-wide little core when the tolerable performance penalty is 1.15x. Compared with Figure 7, little core utilization

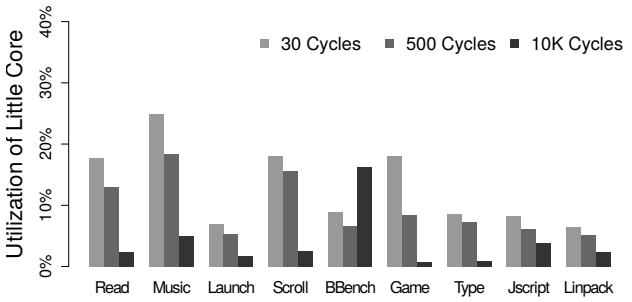

Fig. 12: Little core utilization with 1-wide little core

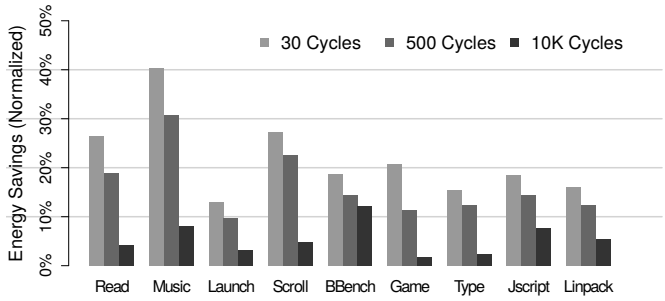

Fig. 13: Energy savings with 1-wide little core

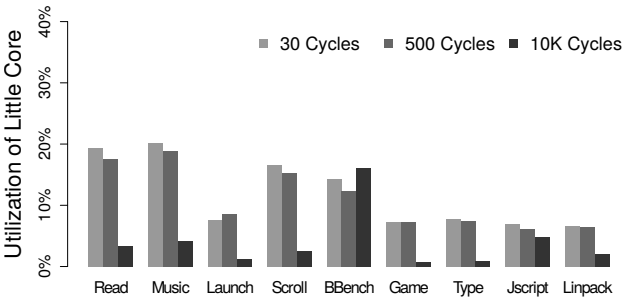

Fig. 14: Little core utilization with 800MHz little core

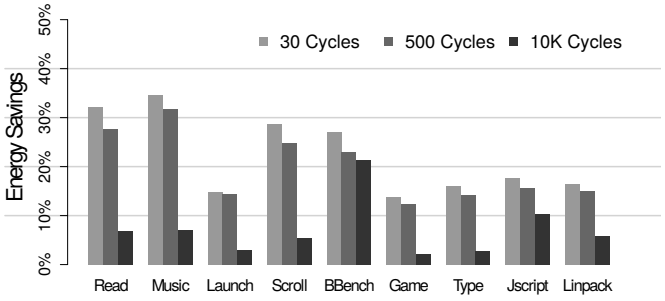

Fig. 15: Energy savings with 800MHz little core

decreases across all AMP design strategies and mobile benchmarks due to the larger performance gap. Figure 13 shows that, although reducing the width reduces the little core's dynamic power from 1.85W to 0.96W, the little core is rarely used and power savings are modest. Thus, the little core must be designed to balance performance and energy savings.

Voltage and Frequency. Existing AMPs provide individual power supplies for big and little cores, which are organized into clusters, allowing the little and big cores to use different clock frequencies. We evaluate the case when the little core's frequency is 800MHz instead of the big's 1GHz.[2] Since frequency differs, we no longer use IPC to determine switch points. Rather, we scale IPC by the frequency difference and measure instructions per second (IPS) such that IPS(big) = IPC(big)/1 and IPS(little) = IPC(little)/1.25, where 1.25 is the slowdown at 800 MHz.

Figure 14 shows little core utilization if 1.15x performance penalty can be tolerated. Reducing little core performance harms its utilization. Although dynamic power falls by half, since voltage can be reduced linearly with frequency, Figure 15 shows that most mobile apps do not benefit from reduced frequency.

In summary, reducing little core performance in return for power savings, either through smaller superscalar width or slower clock frequency, would harm the little core's energy efficiency provided. To maximize energy savings, the little core should provide performance within some competitive range of the big core's so that a switching mechanism can exploit program dynamism at fine granularities.

[2]This difference corresponds to big-little configuration in the ARM Versatile Express development board [6].

VI. Related Work

Heterogeneous multicore architectures, due to their exceptional energy efficiency, have been studied extensively. Previous works have focused on workload assignment [5], [21], [11], scheduling [31], [25], [19], [9], thread performance prediction [32], [35], and design space exploration [34], [12], [22]. The rise of core fusion [16], [18] and adaptive out-of-order cores [23], [26], [17] requires computer architects to rethink heterogeneity and management.

The scheduler for heterogeneous cores must bypass the operating system and make decisions within short time windows. Predicting the next interval's performance is challenging as performance variation increases when the interval length shortens [28]. Researchers have proposed dynamically estimating app performance on another core type [26], [32], [35]. Briefly, the IPC on the other core is estimated with a profiled linear model that takes memory-level parallelism, instruction-level parallelism, and hardware performance counters (IPC, cache hits and misses, branch mispredictions, etc.) as inputs.

Our work is inspired by previous works in heterogeneous processing for mobile platforms. Little Rock [30] prototypes a sensing platform in which a small processor is dedicated to always-on sensor stream processing. GreenDroid [10] uses specialized, low-power cores to accelerate the Android software stack. Zhu and Reddi [36] analyze the loading of webpages on big and little mobile processors, and they propose to predict and schedule the loading of webpages onto heterogeneous cores according to webpage complexity.

VII. Conclusion

We propose a set of interactive mobile benchmarks that captures intra-app diversity, which arises from frequent user inputs. Deploying these benchmarks on Android systems and the gem5 simulator, we explore three asymmetric multiprocessing (AMP) strategies. Strategies with tightly-coupled big and little cores, which share the cache hierarchy, can efficiently utilize little cores and reduce energy by up to 42% for interactive apps. Our benchmarks are open-sourced to help future mobile architecture research [2]. Our evaluation illustrates the rich design space underlying asymmetric multiprocessing for mobile phones.

Acknowledgements

This work is supported by National Science Foundation grants CCF-1149252 (CAREER), CCF-1337215 (XPS-CLCCA), SHF-1527610, and AF-1408784. This work is also supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. Any findings or conclusions expressed in this material are those of the author(s) and do not necessarily reflect those of sponsors.

References

[1] 0xbench, intergrated android benchmark suite by 0xlab. http://code.google.com/p/0xbench/.

[2] Source code. https://github.com/ispass-anonymous/actionbench-.

[3] Variable SMP – A multi-core CPU architecture for low power and high performance. In *NVIDIA White Paper*, 2011.

[4] NVIDIA Tegra 4 family CPU architecture: 4-PLUS-1 quad core. In *NVIDIA White Paper*, 2013.

[5] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating amdahl's law through epi throttling. *SIGARCH Comput. Archit. News*, 2005.

[6] ARM. CoreTile Express A15x2 A7x3. In *ARM Technical Reference Manual*, 2012.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[8] H. Chung, M. Kang, and H. Cho. Heterogeneous multi-processing solution of Exynos 5 Octa with ARM big.LITTLE technology. In *Samsung White Paper*, 2012.

[9] S. Fan, S. Zahedi, and B. Lee. The computational sprinting game. In *ASPLOS*, 2016.

[10] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M.B. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 2011.

[11] M. Guevara, B. Lubin, and B. Lee. Navigating heterogeneous processors with market mechanisms. In *HPCA*, 2013.

[12] M. Guevara, B. Lubin, and B. Lee. Strategies for anticipating risk in heterogeneous system design. In *HPCA*, 2014.

[13] A. Gutierrez, R.G. Dreslinski, T.F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-System Analysis and Characterization of Interactive Smartphone Applications. In *IISWC*, 2011.

[14] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein. Scaling, power and the future of CMOS. In *IEDM*, 2005.

[15] Yongbing Huang, Zhongbin Zha, Mingyu Chen, and Lixin Zhang. Moby: A mobile benchmark suite for architectural simulators. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 45–54, March 2014.

[16] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *ISCA*, 2007.

[17] Khubaib, A. Suleman, M. Hashemi, C. Wilkerson, and Y. Patt. MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *MICRO*, 2012.

[18] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *Micro*, 2007.

[19] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys*, 2010.

[20] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures. In *MICRO*, 2003.

[21] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *SIGARCH Comput. Archit. News*, 2004.

[22] B. Lee and D. Brooks. Illustrative design space studies with microarchitectural regression models. In *HPCA*, 2007.

[23] B. Lee and D. Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In *ASPLOS*, 2008.

[24] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An integrated power, area and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[25] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Supercomputing*, 2007.

[26] A. Lukefahr, S. Padmanabha, R. Das, F. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke. Composite Cores: Pushing heterogeneity into a core. In *MICRO*, 2012.

[27] K. Nose and T. Skurai. Optimization of vdd and vth for low-power and high-speed applications. In *DAC*, 2000.

[28] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Trace based phase prediction for tightly-coupled heterogeneous cores. In *MICRO*, 2013.

[29] Dhinakaran Pandiyan, Shin-Ying Lee, and Carole-Jean Wu. Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite – mobilebench. In *IISWC*, 2013.

[30] B. Priyantha, D. Lymberopoulos, and Jie Liu. Littlerock: Enabling energy-efficient continuous sensing on mobile phones. *Pervasive Computing, IEEE*, 2011.

[31] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys*, 2010.

[32] D. Shelepov and A. Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*, 2008.

[33] Dam Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C.D. Emmons, and N.C. Paver. A structured approach to the simulation, analysis and characterization of smartphone applications. In *IISWC*, 2013.

[34] Kenzo Van Craeynest and Lieven Eeckhout. Understanding fundamental design choices in single-isa heterogeneous multicore architectures. *ACM Trans. Archit. Code Optim.*, 2013.

[35] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *ISCA*, 2012.

[36] Yuhao Zhu and Vijay Janapa Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA*, 2013.

[37] V. Zyuban, J. Friedrich, C. J. Gonzalez, R. Rao, M. D. Brown, M.M. Ziegler, H. Jacobson, S. Islam, S. Chu, P. Kartschoke, G. Fiorenza, M. Boersma, and J.A. Culp. Power optimization methodology for the IBM Power7 microprocessor. *IBM Journal of Research and Development*, 55(3), 2011.