MAESTRO: Autonomous QoS Management for Mobile Applications Under Thermal Constraints

Onur Sahin, Lothar Thiele, *Member, IEEE*, and Ayse K. Coskun, *Senior Member, IEEE*

***Abstract*—Power densities of modern mobile system-on-a-chip designs can quickly exceed the thermal design limits during typical application use such as gaming or Web browsing. Resulting high temperatures lead to frequent thermal throttling and significant loss in quality-of-service (QoS) delivered to users. Thus, a joint consideration of thermal constraints and QoS requirements is essential to maximize the overall user experience. Prior techniques either rely on users to determine the best tradeoff point between QoS and temperature, or greedily utilize the thermal headroom to maximize performance, causing QoS to drop below user tolerable levels over extended durations of use. This paper introduces the MAESTRO framework to automatically manage QoS at runtime depending on application characteristics and thermal constraints. MAESTRO builds on the observation that increased temperatures can be tolerated for applications with bursty compute patterns due to idle periods between activities, while causing large QoS degradations for long-running applications with continuous computations. MAESTRO: 1) detects such continuous computations that are susceptible to throttling; 2) proactively finds a QoS level to balance user experience and temperature; and 3) performs closed-loop DVFS and thermally efficient thread mapping to meet the target QoS on a heterogeneous multicore CPU. Such application-adaptive control of QoS-temperature tradeoffs allows MAESTRO to *sustain a target QoS level within a user tolerable range for longer durations* without sacrificing the performance of latency-sensitive bursty computations. Evaluations on a real system prototype validates MAESTRO's ability to accurately detect potential throttling-induced QoS degradations and demonstrates 41% to 6.7× longer durations of sustained QoS compared to state-of-the-art for a set of mobile applications.**

***Index Terms*—Heterogeneous multicore, mobile devices, power management, thermal management.**

I. INTRODUCTION

STATE-OF-THE-ART mobile system-on-chips (SoC) integrate high-performance CPUs and GPUs to enhance user experience. As the applications grow in complexity and utilize the peak processing capabilities of the underlying hardware, power dissipations can easily reach above the thermal design limits. For instance, recent SoCs can consume up to 3× higher power than their thermal design power and chip temperatures can quickly reach critical thresholds while executing typical mobile applications [23], [41], [43]. Thermal limitations present a major roadblock for increasing the compute capabilities of mobile CPU and GPUs toward providing higher quality-of-service (QoS)[1] for mobile applications. In fact, frequent invocations of thermal throttling mechanisms triggered by temperature violations can significantly degrade QoS and cause user dissatisfaction [8], [9].

Built-in policies in modern mobile systems (e.g., dynamic voltage and frequency scaling (DVFS) governor, task scheduler, etc.) greedily maximize performance under increased computational demand by selecting higher DVFS states or higher performance CPU cores. Such an approach of always maximizing performance has various drawbacks that can substantially hurt user experience. First, performance can exceed human perceptible levels when QoS requirements for an application are not considered, causing unnecessarily high power consumption levels that will accumulate more heat on system components and increase throttling. Second, thermal headroom will be prematurely exhausted, further magnifying the performance loss due to thermal throttling over extended durations of application use (e.g., as in gaming, streaming). Therefore, there exists a need for *QoS-centric thermal management approaches* that can, as opposed to greedily increasing performance, *proactively find a good tradeoff between QoS and temperature to a sustain satisfactory user experience* over time.

We face fundamental challenges when deploying such a scheme for managing QoS-temperature tradeoffs in mobile platforms: how should one determine when to apply such a tradeoff and the appropriate amount of QoS scaling? While completely neglecting QoS tradeoffs can result in large throttling-induced QoS degradation over a long term, prematurely enforcing a tradeoff can result in undesirable performance losses on applications where throttling would have little/no effect or where the maximum QoS is demanded by the users. Ideally, the policies should tradeoff QoS only on cases where large QoS degradations are expected over the extended use. In addition, the scaling of QoS should still occur within a user tolerable range in order not to deem the application unusable. Currently, there exists no mechanism to address these objectives all together. Existing methods incur practical limitations as they rely solely on users to manage QoS [43], [44] or seek to achieve sustained performance with low-power operation modes in a QoS- and application-agnostic manner, which can result in

Manuscript received December 19, 2017; revised March 21, 2018; accepted May 23, 2018. Date of publication July 12, 2018; date of current version July 17, 2019. This paper was recommended by Associate Editor H. Li. *(Corresponding author: Onur Sahin.)*

O. Sahin and A. K. Coskun are with the Electrical and Computer Engineering Department, Boston University, Boston, MA 02215 USA (e-mail: sahin@bu.edu; acoskun@bu.edu).

L. Thiele is with the Computer Engineering and Networks Laboratory, ETH Zürich, 8092 Zürich, Switzerland (e-mail: thiele@tik.ee.ethz.ch).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2855180

[1]QoS, in this paper, refers to a metric used to quantify the performance experienced by the user (i.e., user experience). We use various QoS metrics such as frames-per-second (FPS) and response latency (Section II).

unacceptably low QoS (e.g., Android's Sustained Performance API [6]).

This paper introduces the MAESTRO framework for achieving autonomous and application-aware QoS tradeoffs. MAESTRO builds upon a novel insight on the relation between the computation characteristics of mobile applications (i.e., bursty versus throughput-oriented) and the QoS impact of thermal throttling. While throughput-oriented mobile applications (e.g., gaming and video processing) may suffer from long-term throttling due to continuous computations and require QoS tradeoffs to sustain an acceptable throughput (e.g., FPS), some applications only generate short bursts of computations in response to user interactions. Web browsing and many interactive Android apps (e.g., news reading, social networking, messaging, and document reading) are examples of such bursty applications, where latency of the computations is the main factor that impacts the user perceived QoS [17]. Despite high power densities and increased temperatures, such applications can tolerate increased temperatures due to relatively short duration of activities and idle periods between user interactions. MAESTRO classifies the applications at runtime according to their computation characteristics to distinguish throttling-susceptible continuous computations from latency-sensitive bursty tasks, and manages QoS accordingly.

MAESTRO tracks the statistical features of an application's power profile at runtime to infer when large thermally induced QoS degradations on continuous computations are likely to occur (Section IV-A). Upon detecting such a computation phase, MAESTRO proactively scales the QoS down from the maximum to a level that can be sustained for a longer time (Section IV-B). Otherwise, MAESTRO uses the existing CPU governors to maximize QoS. To maintain the QoS at a *just enough* level to meet the determined QoS targets, MAESTRO incorporates a closed-loop and thermally efficient QoS control strategy (QScale) proposed in our earlier work [43] (Section IV-C). While QScale [43] relies on external sources (e.g., users) to manage QoS decisions, this paper introduces a novel runtime framework for thermally aware and autonomous QoS tradeoffs to achieve sustainable performance.

Overall, our contributions can be summarized as follows.

1) *Application-Specific Thermal Management:* We show that taking into account the bursty or throughput-oriented nature of computations in mobile applications is necessary to project throttling-induced QoS degradations that can significantly impact user experience.
2) MAESTRO *for Sustainable QoS:* We design MAESTRO, which automatically reasons about the susceptibility of an application to thermal throttling and proactively manages QoS to increase durations of sustained performance.
3) *Evaluations on Real Systems:* We implement and evaluate MAESTRO on a real-life mobile platform. MAESTRO can accurately identify when throttling-susceptible long computations occur during runtime. Our experiments indicate 41% to 6.7× longer durations of sustainable QoS.

The rest of this paper starts with a description of our experimental setup and methodology in Section II. In Section III, we explain and experimentally demonstrate the motivation for MAESTRO by illustrating the need for an application-aware QoS management strategy. Section IV presents the MAESTRO policy as well as providing an overview of the core concepts of QScale [43] proposed in our earlier work. In Section V, we evaluate MAESTRO via implementation on a real-life mobile development board. Section VI provides a summary of relevant work in the field and highlights the key distinguishing aspects of this paper. Section VII concludes this paper.

II. Experimental Setup

This section presents the hardware testbed and applications we use in our experimental evaluation and describes our data monitoring/collection methodology.

A. Experimental Platform

All of our measurements and evaluations are based on real-life experiments on a contemporary mobile hardware. We use an Odroid-XU3 mobile development platform that comprises of the Samsung Exynos 5422 SoC (which powers Samsung Galaxy S5 smartphone), implementing a big.LITTLE heterogeneous CPU architecture [2] with quad-core big (A15) and little (A7) CPU clusters. The A15 is a high performance/power multi-issue out-of-order processor and A7 is a low performance/power core with simple eight-stage in-order pipeline [2]. The A15 core supports nine frequency levels from 1.2 to 2 GHz while the A7 core operates on five frequency levels between 1 and 1.4 GHz. All the cores within a cluster share the same voltage/frequency domain. The Exynos 5422 SoC also integrates Mali-T628 GPU, which supports six frequency levels ranging from 177 to 543 MHz. A built-in mechanism scales the GPU frequency based on utilization. The board runs Android 4.4 KitKat as the OS. While we cannot use recent Android versions due to unavailability of system images for our system, we incorporate the Sustained Performance API feature [6] of Android 7 for evaluation (Section V) due to its direct relevance to this paper.

B. Measurement Methodology

The Odroid-XU3 platform is equipped with on-board sense resistors and a Texas Instruments INA231 power monitoring unit that allows for measuring power consumptions of the A15 and A7 clusters, the GPU, and the memory individually over the I^2C bus. Temperatures of each of the four big cores and the GPU can be sampled at a 1 °C resolution through the `sysfs` entries provided for on-chip thermal sensors. We collect power and temperature data in 5-ms intervals. FPS is measured by querying the logs generated by the *SurfaceFlinger* Android system service. We measure the latency of bursty computations by the length of time between the rising and falling edge of the burst observable in a given power profile.

C. Applications

We broadly classify mobile applications into two classes as latency-sensitive bursty (e.g., browsing and interactive apps) and throughput-oriented workloads (e.g., games and streaming) according to their computation characteristics [45], [53]. Our experimental setup covers applications with both throughput-oriented and latency-sensitive bursty computations. Table I summarizes these applications along with brief descriptions of the tasks performed within each application.

Adobe PDF Viewer and *Google Maps* represent two important classes of mobile applications, document reading, and navigation. *Caman.js* [3] is an online image editing application

TABLE I
SUMMARY OF APPLICATIONS

Application	Description	QoS Metric
PDF Viewer	Open a PDF, read, zoom in/out figures	Latency
Google Maps	Search location, move across the map	Latency
Caman.js [3]	Apply different filters on an image	Latency
Bodytrack	Process image files	Heartbeats/sec
Edge of Tomorrow	Loading, menu selection and gaming	FPS
Aquarium [1]	Watch online animation	FPS
Rain [7]	Watch online animation	FPS
Rock Player	Open and play a video file	FPS

that we execute within *Chrome* Web browser. These latency-sensitive applications generate bursts of CPU loads upon user inputs from the GUI. The latency of a processing event is the main factor impacting user experience [17], [51] and is chosen as the QoS metric. While there is not always a strict deadline for processing such computations, the longer latencies increase user dissatisfaction.

The rest of the applications are dominated by continuous computations. We run *Aquarium* [1] and *Rain* [7] applications within *Chrome* browser to play online WebGL animations. *Edge of Tomorrow* gaming application is also representative of throughput-oriented mobile workloads. We also use *Rock Player* video player application to continuously play a 1-min HD video and loop the video to experiment with longer durations. In such applications that generate a stream of computations, user experience is manifested in event throughput and commonly measured using FPS [41], [53] as in this paper. The FPS metric captures the number of frames that meet the frame processing deadline (i.e., 18 ms at 60 frames/s). Finally, we run the *bodytrack* computer vision application from the PARSEC suite [12] where we monitor the frame-rate (or heartbeats/s) by instrumenting the application with the Heartbeats framework [27]. We configure bodytrack to run with the same number of threads with the number of CPU cores in our system (i.e., eight cores). In order to automate the execution of interactive applications and achieve reproducible results, we use RERAN, which is a GUI-based and timing-sensitive record and replay tool [20].

We also write custom CPU and GPU microbenchmarks for the offline thermal coupling characterization. Our GPU microbenchmark is an OpenCL program that repeatedly offloads a matrix multiplication kernel to GPU. Our CPU microbenchmark continuously performs floating-point multiplications.

During evaluation, we run each throughput-oriented application for a sufficiently long duration (i.e., 12 min) to eventually cause throttling and quantify the exact duration of sustained QoS. Bursty applications are run for 60–70 s, which is a typical duration for interactive sessions [18].

D. Power Management and Throttling

Our platform uses an external fan for temperature control, which is not a viable solution for commercial devices. Thus, we implement a thermal throttling policy that reactively increments/decrements the *maximum allowed DVFS state*[2] of big cores every second if the maximum temperature is lower/higher than 80 °C. 80 °C is a typical thermal set-point used in commercial platforms [41], [50]. By changing the maximum DVFS levels, this throttling mechanism forces the DVFS policies to use lower frequencies without disabling their operation. In case a thermal emergency still exists at the lowest big core frequency, the workload is migrated to little cores using the *sched_setaffinity* interface in the Linux scheduler.

[2]Maximum allowed DVFS state can be altered by modifying the `scaling_max_freq` sysfs entry provided by the cpufreq interface [5].

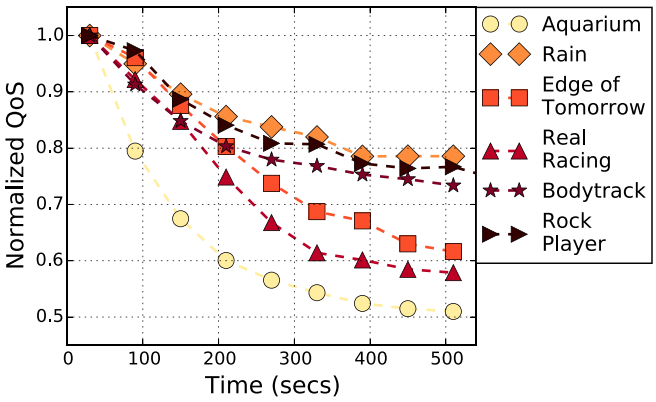

Fig. 1. QoS degradation due to thermal throttling over time.

The default DVFS policy in our platform is the *Interactive governor* [5], which is also used in most Android devices. This governor scales the CPU frequency to the maximum allowed level if the utilization is higher than a threshold. Once scaled to the highest, CPU frequency is not scaled down for at least 20 ms to maximize responsiveness. The baseline heterogeneous multiprocessing (HMP) scheduler [4] migrates an active task to a big core if its weighted average CPU load exceeds an *up_threshold*. Migration to little cores occurs similarly when the load is less than a *down_threshold*. We collectively refer to *Interactive governor* and HMP scheduler pair as "default" Android management throughout this text. MAESTRO uses *cpufreq* and *sched_setaffinity* interfaces to control the frequency and the thread mappings for an application. We bind the policy to a dedicated little core using the `taskset` utility.

III. MOTIVATION: APPLICATION-AWARE QOS VERSUS TEMPERATURE TRADEOFFS

This section describes the problem we focus on in this paper and illustrates the main insight and motivation behind the proposed MAESTRO approach.

A. (Un)Sustainable Performance

Many mobile applications, such as video playing and gaming, perform a continuous stream of computations (e.g., in form of frames) and can be run for durations in the order of minutes. Thus, user-perceived QoS is manifested in terms of frame throughput [13]. For such applications, greedily boosting QoS can lead to increasingly aggressive throttling over time, causing high QoS to be maintained only for a brief duration at the cost of severe QoS losses over the application use. Fig. 1 illustrates this problem as we run various throughput-oriented applications for an 8–10 min duration using default scheduling and DVFS policies on our experimental platform. Significant QoS degradations occur due thermal throttling over extended durations, reaching up to as much as 48% QoS loss.

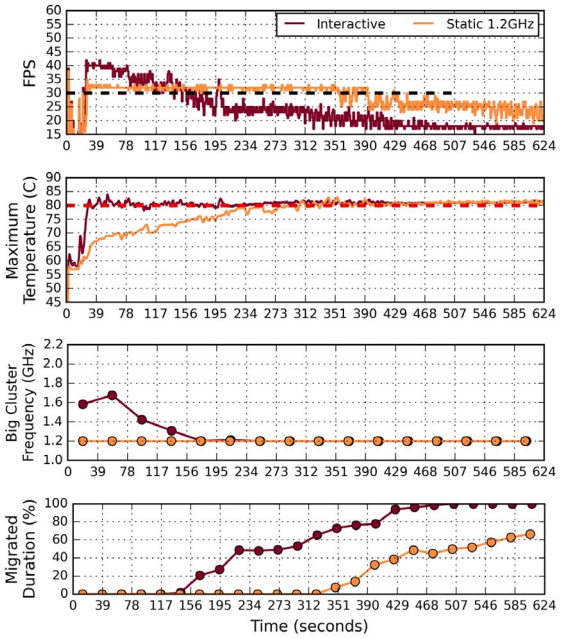

Fig. 2. QoS-temperature tradeoff for sustainable QoS while running Aquarium application with default Interactive governor and with static 1.2-GHz frequency. Using the 1.2-GHz setting allows to sustain a target QoS level (e.g., 30 frames/s) for longer durations. Migrated duration (%) corresponds to percentage of time the workload was migrated to little cluster due to thermal throttling.

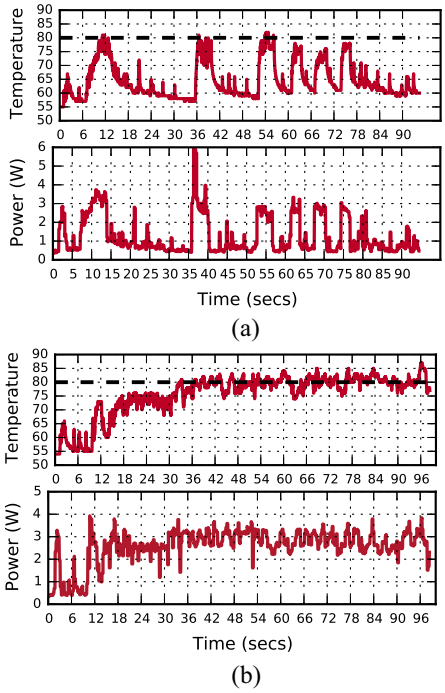

Fig. 3. Distinct power and thermal profiles of a bursty application and a throughput-oriented graphics application. (a) *Adobe PDF Reader* application. (b) *Rain* WebGL animation running in Chrome Web browser. *Adobe PDF* application consists of bursts of computations such as zoomed-in/out or text search as generated upon user input from GUI. Such intermittent nature of computations is widely observable in the power profile as well. On the other hand, continuous frame-based computations after the browser launch (t = 18 s) in *Rain* application causes a relatively more steady power profile.

B. Trading Off Short-Term QoS for Temperature

For the applications where maximum QoS cannot be maintained due to increased thermal throttling over time, *trading off short-term QoS helps increase the duration that a target just enough QoS level is sustained*. Fig. 2 illustrates this insight. We run a graphics animation application (*Aquarium*) for 8 min using both Android's default *Interactive* DVFS governor as well as when the maximum frequency of big cores are statically fixed to 1.2 GHz from the default 2.0-GHz setting. It should be noted that, in both cases, the throttling policy (Section II) is intact and forces the use of lower frequencies when the maximum temperature exceeds 80 °C. With the baseline *Interactive governor*, the maximum QoS (i.e., FPS) can be maintained for a brief 40-s duration, after which the CPU is set to operate at lower DVFS levels by the throttling policy as higher frequencies incur thermal violations. Frequent migrations to the little cluster occur eventually as the 1.2-GHz setting becomes insufficient to reduce the temperature. Trading off short-term QoS with the 1.2-GHz setting slows down the heating and maintains a minimum FPS of 30, which is pointed by prior work as the lowest user tolerable FPS for gaming applications [13], for 5 min (∼2x longer as compared to default *Interactive*) until the thermal headroom is exhausted.

C. Need for Application Awareness

While QoS-temperature tradeoffs are beneficial in terms of extended sustained QoS durations, it is nontrivial to decide when applying such a tradeoff would bring benefits. This is because of several reasons. First, a computation initiated for an activity may exhaust thermal headroom but may be short-lived and cause only a brief duration of throttling (e.g., a few seconds). For such short and bursty computations, minimizing latency would be more desirable from the user's perspective than maintaining a sustainable throughput [53]. Naively switching to a lower performance setting (e.g., upon thermal violation) for sustainable QoS will lead to unnecessary QoS loss and increase user-perceived latency for bursty tasks. Second, even for the applications that do perform long-running computations (e.g., minutes), throttling mechanisms may have little or no impact on QoS for relatively low power applications. Thus, conservatively applying a QoS tradeoff will cause an unnecessary performance loss.

Consider the *Adobe PDF reader* and *Rain* graphics animation applications that present disparate computation patterns. Fig. 3 shows the power and thermal profiles for these two applications as they run under default Android management. We leave the thermal control policy enabled to prevent thermal runaway. *PDF reader* application [Fig. 3(a)] generates short bursts of intense computations upon user input (e.g., opening a PDF, text search). Initiation and ending of computations can be inferred from the power profile. Due to short-lived nature of computations and idleness between the user inputs, temperature can quickly decrease from the critical level. *For such applications, it is tolerable to allow the application to exhaust thermal headroom and achieve maximum QoS*. *Rain* application, on the other hand, performs continuous computations for frame processing after being launched in the browser at t = 15 s. This continuous load causes a relatively stable power consumption (∼3 W) and consistent increase in temperature. As shown in Fig. 1, throttling incurs significant QoS loss due to continuous thermal violations and more aggressive

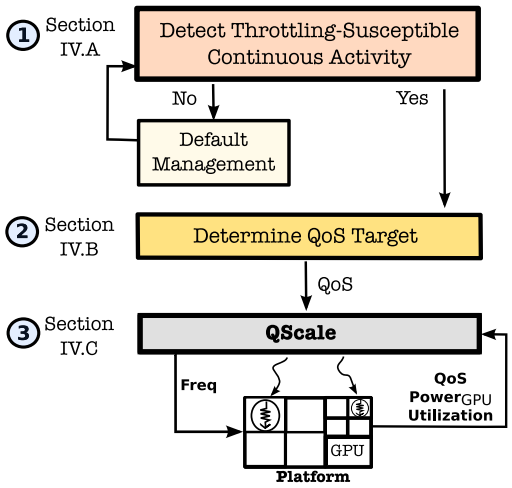

Fig. 4. Overview of MAESTRO.

throttling. Thus, we argue that, to be able to effectively apply QoS management for sustainable user experience only when it would be desirable by a user, policies should be cognizant of both application behavior and system thermal constraints.

IV. MAESTRO: AUTONOMOUS QoS TRADEOFFS FOR SUSTAINABLE PERFORMANCE

This section describes the proposed MAESTRO framework for dynamically managing QoS levels in mobile applications. Fig. 4 gives an overview of our proposed technique that is composed of three main components. The online detection policy (Section IV-A) identifies long-running CPU intensive phases that are prone to throttling and QoS degradation, and selectively activates QScale for sustainable performance or uses default Android management for high QoS. If activated, depending on how much the application is likely to suffer from throttling, target QoS level is scaled accordingly and given as an input to QScale [43] (Section IV-B). QScale (Section IV-C) uses the offline generated criticality information on various applications and monitors CPU-GPU thermal coupling dynamics to determine how to map threads on a heterogeneous CPU with the aim of minimizing temperature. Closed-loop DVFS control within QScale ensures dynamic adaptation to changes in workload as well as QoS requirements.

A. Proactive Detection of Throttling-Induced QoS Loss

The goal of our online detection policy is to identify long and continuous computations that are likely to cause severe QoS degradations due to throttling (e.g., *Rain* application in Fig. 3(b)). Such detection allows us to take proactive actions before the system heats up aggressively over time.

We devise a simple yet effective online policy (Fig. 5) that infers the thermal behavior of a mobile application based on inherently distinct patterns in the power profiles of bursty and throughput-oriented computation phases. We use power due to its direct relevance to temperature. Due to continuous computations in the throughput-oriented periods, their power will have a *relatively more stable* profile as compared to bursty compute phases that exhibit intermittent power profile due to idle periods in between the computations (see example in Fig. 3). We track mean (μ) and standard deviation (σ) using a

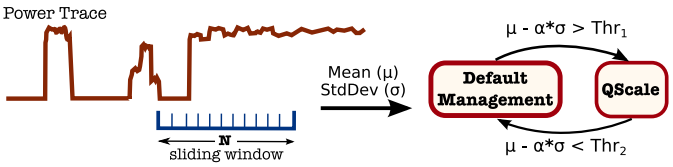

Fig. 5. Sliding window-based online detection policy.

sliding window of recent power samples (big+LITTLE+GPU power) to capture characteristics of the power profile. As we seek to identify the phases with *stably high* power that are likely to suffer from QoS loss, we combine mean and deviation into an activation function (f_{act}) as $\mu - \alpha * \sigma$ (α is a constant scaling factor) to quantifiably identify such phases. Our policy checks whether the value of f_{act} is greater than a certain threshold and, if true, actives QScale for sustained performance. Disabling QScale and resorting to default Android policy for high QoS occurs similarly by comparing the value of f_{act} to a lower threshold. Higher deviation in the power profile of bursty phases reduces the value of f_{act} and allows to prevent false positives (i.e., activating QScale during bursty periods). By giving a different weight to mean and deviation via the scaling factor (α), we are able to tune our policy to distinguish bursty computations while accounting for potential variations that can occur during continuous computations (e.g., differences in the subsequent frames being processed).

1) Tuning Policy Parameters: Since the behavior of the proposed policy would depend on its parameters (i.e., N, Thr_1, Thr_2, and α), we describe our intuition behind selection of these parameters to make effective use of our technique. The length of the sliding windows needs to be sufficiently long to capture the idleness following the compute bursts. This is necessary to distinguish continuous computations from intermittent activity bursts. Thus, we use 10 s window size as vast majority of bursty activities finish within 10 s [52], [53]. While the window size can be conservatively increased, that will add unnecessary delay into the detection. We experiment at different DVFS levels to determine power levels that cannot be sustained over extended durations (i.e., violates 80 °C thermal limit), and use this level to set Thr_1 (i.e., 2 W). Higher mean power values signal potential future throttling and QoS loss. Once Thr_1 and N is set, we use our bursty workloads to tune the value of α (i.e., 0.8) by giving higher weight to deviation in f_{act} until the false positive cases are eliminated. We set Thr_2 to 1.2 W, which is sufficiently lower than Thr_1 to avoid oscillatory behavior.

B. Determining QoS Targets

Once MAESTRO detects a continuous intensive computation, it activates QScale and supplies a target QoS level to be maintained. Many heuristics can be applied for making this QoS tradeoff but no *golden rule* exists as the suitability of a particular QoS level is subject to preferences of a particular user in terms of the performance needs [51] as well as the duration of application use. In our implementation, once the QScale is activated to increase sustained durations, we tradeoff QoS by scaling down from its maximum level in accordance with application's mean power level over the sliding window described in Section IV-A. This mechanism is illustrated in Fig. 6. Our intuitive rationale is that, as the power increases, choosing high QoS settings will quickly exhaust the thermal

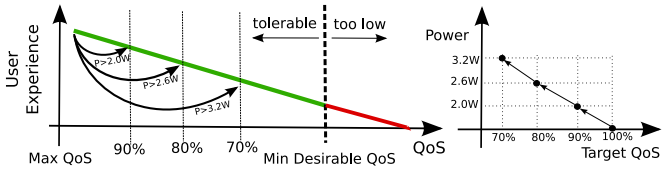

Fig. 6. QoS tradeoff and determining QoS targets for QScale.

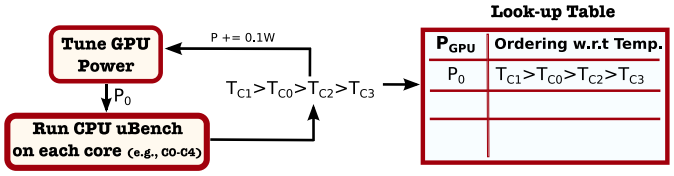

Fig. 7. Offline CPU-GPU thermal coupling characterization to derive a runtime thermally aware core allocation method.

headroom and bring limited or no benefit in terms of extended sustained QoS. Thus, a larger QoS tradeoff within a tolerable range is needed for applications with higher power profile. To determine the ratio in which QoS is scaled down from the 100% when mean power exceeds the 2 W activation threshold of QScale, we consider the QoS level needed (i.e., 70%) for ensuring at least 2 min of duration without throttling for a typical possible high power application (i.e., >3.3 W by *Bodytrack* in our example application set). We simply proportionally scale QoS within the 2–3.3 W range by choosing a 10% lower target QoS rate for every 600 mW as illustrated in Fig. 6. Aiming for longer/shorter than 2 min minimum target sustained duration translates into larger/smaller steps in the tradeoff, approaching/distancing from the undesirable QoS region illustrated in Fig. 6. We considered 30 and 24 frames/s as the minimum desirable QoS for 3-D graphics and video playing scenarios, respectively.

While we consider the problem of autonomously managing QoS without requiring user intervention, user feedback can still be integrated with MAESTRO to provide user-specific hints. Specifically, if the user requires a higher QoS than provided, the minimum desirable QoS can be raised. This will shift the target QoS upward. If the user provides a hint that current QoS is too high, the maximum QoS parameter can be lowered which will cause MAESTRO to choose a lower QoS.

C. Online Thermally Efficient QoS Control

In order to meet the target QoS levels, we adopt QScale from our earlier work [43]. This section serves as an overview of the core components of QScale, which comprises an offline characterization phase and an online feedback control policy. In the offline phase, we: 1) characterize the CPU-GPU thermal coupling via a set of microbenchmarks and derive a lightweight heuristic for *runtime thermal coupling aware core allocation* and 2) identify the few number of *critical threads* within each application that determine overall QoS. Online policy identifies and reserves thermally efficient big cores during runtime for executing the critical threads while leveraging low-power little cores for other threads, and performs a closed-loop DVFS control to precisely meet QoS targets.

1) CPU-GPU Thermal Coupling Aware Thread Mapping: Due to tight integration of power-hungry CPUs and GPUs on a single-chip, thermal coupling is inevitable. Our measurements indicate as much as 25 °C increase in CPU temperature due to GPU activity. A major novel aspect of this paper is to show the application dependence of the cores that provide the most thermally efficient operation under thermal coupling.

In order to infer the most thermally efficient cores to allocate under varying thermal coupling scenarios, we derive an offline characterization step as highlighted in Fig. 7. By stressing the GPU at varying levels and running the CPU microbenchmark on each core (e.g., for 1 min), we determine the ordering in which the cores heat from the highest to the lowest. We use the microbenchmarks described in Section II. Combining the GPU power levels that provide the same ordering, we obtain the simple threshold-based policy shown in Table II. During runtime, the policy allocates the cores that will provide the lowest temperature based on application's GPU usage. In our experimental platform, Core3[3] always provides the lowest temperature while the thermal-efficiency of the other cores depend on the GPU power. Due to close proximity to GPU, Core0 results in the worst peak temperature when the GPU power is high (i.e., $\text{Thr}_2 \leq P_{\text{GPU}}$) and is allocated as the last resort. On the other hand, GPU acts as a heat spreader when its power is low (i.e., $P_{\text{GPU}} < \text{Thr}_1$) and Core0 achieves the lowest temperature. Our methodology is *black-box*, not requiring any floorplan or packaging details.

TABLE II
OUR THRESHOLD-BASED POLICY FOR ALLOCATING THERMALLY EFFICIENT CORES UNDER CPU-GPU THERMAL COUPLING. $\text{Thr}_1 = 0.25$ W AND $\text{Thr}_2 = 1.2$ W

Condition	Order of Big Core Allocation
$Thr_2 \leq P_{GPU}$	Core3, Core1, Core2, Core0
$Thr_1 \leq P_{GPU} < Thr_2$	Core3, Core1, Core0, Core2
$P_{GPU} < Thr_1$	Core3, Core0, Core1, Core2

2) Criticality Aware big.LITTLE Scheduling: Current schedulers for big.LITTLE systems [4] use the short-term load history of each thread to determine the core type. Instead, we propose to guide scheduling by leveraging power-hungry big cores only for the threads that are critical for the user experience. Our novel observation is that there exist relatively (as compared to number of cores) few number of *QoS-critical threads* that determine overall QoS, which is in alignment with the prior work that has shown limited parallelism in mobile applications [19], [45]. We determine such critical threads via a simple offline characterization process.

During the offline characterization, in our earlier work [43], we have exhaustively explored the QoS contribution of each thread when executed on a high-performance big core by assigning all threads to little cluster first and moving to the big cores one at a time. A few threads provide distinct increase in QoS once assigned to big cores. On a per-application basis, we record several identifier information (i.e., thread name and relative order of the thread id in parent's process list) for each of these critical threads to be used by the runtime control policy. Exhaustively exploring QoS contribution of all threads, however, increases the characterization time as the number of threads can be large (e.g., 50–100). Indeed, vast majority of the threads are short lived and largely stay in `sleep` state after the application launch [24]. For instance, Fig. 8 shows the

[3]Core0–Core3 correspond to cpu4–cpu7 under the `/sys/devices/system/cpu/` path in the Odroid-XU3 filesystem.

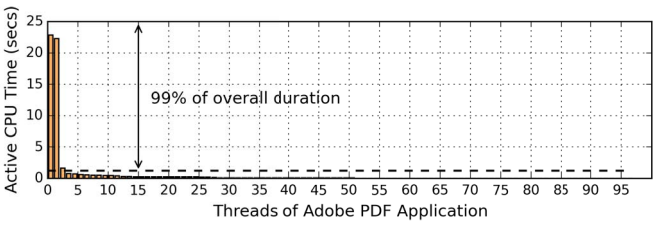

Fig. 8. CPU time distribution across the threads of the *Adobe PDF Reader* application demonstrating that most threads have little active time on the CPU. Dashed line corresponds to 1% of the overall execution time.

Application	QScale Enabled?
Aquarium	✓
Rain	✓
Rock Player	✓
Edge of Tomorrow	✓
Bodytrack	✓
Maps	✗
Adobe PDF	✗
Caman	✗

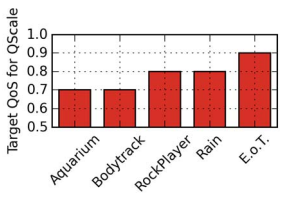

Fig. 9. Policy selection (left) and QoS targets (right) determined by MAESTRO for the applications where MAESTRO detects a continuous throttling-prone computation. Policy selection and QoS setting are based on the methods described in Sections IV-A and IV-B, respectively. MAESTRO assigns lower QoS targets for the applications that exhibit high power profile and that are likely to suffer from larger QoS loss. Maximum QoS is 1 HB/s for *Bodytrack*, and 45, 30, 53, and 55 frames/s for *Aquarium*, *RockPlayer*, *Rain*, and *Edge of Tomorrow*, respectively.

time each thread is actively utilizing CPU for the *PDF Reader application* during a typical usage session of 70 s [18]. Overall active CPU time of the application is dominated by a small number of threads. We exploit this observation to reduce the characterization time by excluding such mostly idle threads from the characterization step. We empirically determined to exclude the threads that are active for less than 1% of the overall execution time. Since energy and thermal footprint of such threads would be small, we do not investigate any further gains that could be realized by choosing the right core type for these threads. Overall, the number of critical threads per application are identified as 5 for *bodytrack*, 2 for *Aquarium* and *Rain*, 1 for *Caman.js* and *Adobe PDF Reader*, and 0 for the *Google Maps*. We leave the comparative evaluation of criticality-aware scheduling and baseline HMP scheduler to Section V and proceed with a brief description of our closed-loop runtime control policy.

3) Closed-Loop Runtime Control: QScale's runtime policy, which is invoked when MAESTRO detects a long running throttling-prone phase, consumes the offline generated thread-criticality and thermal coupling information to meet the target QoS levels while minimizing temperature. During runtime, the policy monitors GPU power and uses the policy described in Table II to determine thermally efficient cores for executing critical threads. To precisely meet the target QoS levels, a feedback DVFS controller is designed.

QScale's runtime controller initially assigns the threads to little cluster and, at every 1 s intervals, allocates the threads across big and LITTLE. When the number of critical threads is less than 4 (which is the case for all applications except *bodytrack*), there exists opportunity to lower temperature by making thermally aware core allocation. In this case, we sequentially bind the critical thread with the highest CPU usage to the next most thermally efficient core available by querying Table II. Otherwise, the critical threads are allocated to the whole big cluster and default Linux load balancer is used for task mappings within the cluster.

Once the threads are allocated, the closed-loop controller determines the DVFS state for the next interval that will meet the target QoS. The formulation of our proportional integral controller is shown in (1)

$$u[k] = u[k-1] + \frac{e[k](1-p)}{Q_{\max}}. \tag{1}$$

The error term ($e[k]$) is simply the current offset from the target QoS. The controller output ($[0, 1]$) is scaled proportionally to the DVFS range. The value of the pole (p) value should be in range $[0, 1)$ to ensure stability and avoid oscillatory behavior [25]. This parameter also allows to tradeoff controller's robustness for responsiveness [25] and a smaller value increases the controller's response to workload variations. We empirically determine the value of p to be 0.4 on our system.

V. EVALUATION

This section provides a detailed real-system evaluation of the proposed MAESTRO policy. Our main objective is to assess MAESTRO's ability to provide extended durations of sustained QoS by proactively identifying the throttling-susceptible continuous computations and autonomously making QoS tradeoffs. We also craft specific experiments to evaluate the adaptive runtime behavior of MAESTRO and carefully study any overhead that could lead to performance degradations to assess MAESTRO's suitability as a runtime management solution. We refer to the built-in *Interactive* governor and *HMP* scheduler pair simply as default Android management throughout this section and provide comparisons against MAESTRO that employs closed-loop DVFS and criticality-driven scheduling. The runtime policy within MAESTRO, which performs the QoS control via DVFS and criticality-driven scheduling, is codenamed and referred to as QScale.

A. Evaluation Methodology

This section states the methodology adopted while conducting the experiments and evaluating the outcomes. We exercise the bursty-dominated *Adobe PDF*, *Caman*, and *Google Maps* applications for 60–70 s, which is the typical length of a user session for many interactive mobile applications [18]. We run each throughput-oriented *Aquarium*, *Rain*, *Edge of Tomorrow*, *Rock Player*, and *Bodytrack* applications for 10 min. This duration is sufficiently large for each application to trigger throttling and allows us to quantify the exact sustained duration before the thermal headroom is fully exhausted. The sustained duration for MAESTRO is the duration before thermal throttling starts to force lower DVFS settings, beyond which QScale can no longer deliver target QoS. For the default Android management, we simply report the execution time that QoS was above the target level as the sustained QoS duration. To ensure fidelity during temperature measurements, we cool the SoC to the initial idle temperature level (i.e., 59 °C) using an integrated fan mounted on top of the chip package and leave the platform idle for 12 min before each experiment.

B. Extending Sustained QoS With MAESTRO

This section evaluates MAESTRO's selective QoS tradeoff mechanism (Sections IV-A and IV-B) as well as quantifying

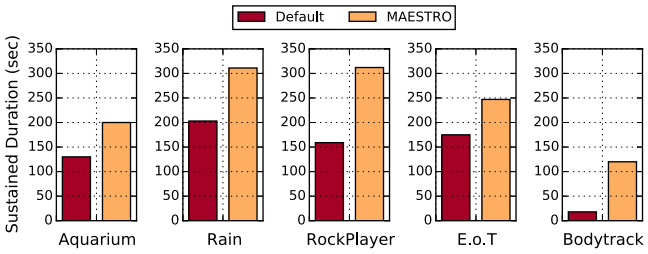

Fig. 10. Sustained durations achieved by MAESTRO and default Android management for the QoS targets specified in Fig. 9.

potential improvements in sustained QoS durations achieved on CPU intensive throughput-oriented applications. The table given in Fig. 9 shows the policy selection of MAESTRO across different applications. MAESTRO successfully detects the continuous and throttling-prone computations in *Aquarium*, *Rain*, *Edge of Tomorrow*, *Rock Player*, and *Bodytrack* applications and activates QScale in all cases with the target QoS levels shown also in Fig. 9. Once activated, QScale maintains the QoS at these target levels during the sustained duration. *Aquarium* and *Bodytrack* are two applications with distinctly high power consumption (>3.5 W) during the throughput-oriented phase and, thus, are assigned a lower target QoS (i.e., 70% of the maximum). MAESTRO recognizes the latency-sensitive bursty computation patterns in *Pdf*, *Caman*, and *Maps* applications and does not interfere with the default Android management, allowing users to enjoy high QoS without sacrificing latency. Such adaptive policy selection capability allows MAESTRO to make QoS tradeoffs and sustained performance optimization only when necessary.

Fig. 10 provides an evaluation of the sustained QoS durations achieved by using MAESTRO and default Android management for the QoS targets described in Fig. 9. *Bodytrack* and *Aquarium* are the cases with the lowest duration where target QoS levels are met, using both MAESTRO and baseline. Despite the selection of a low QoS target by MAESTRO (i.e., 70% of the maximum) for these two applications, temperatures still quickly elevate to critical 80 °C level due to high CPU activity and power. Target QoS is violated as throttling forces CPU to operate at lower DVFS levels, shrinking the duration of sustained QoS in these two cases.

MAESTRO provides 41%, 53%, and 54% longer durations where QoS targets are met for *Edge of Tomorrow*, *Rain*, and *Aquarium* applications, respectively. Such an improvement is achieved by both thermal coupling aware assignment of threads as well as by proactively identifying the throttling-induced large QoS degradations to make the necessary QoS-temperature tradeoffs. The benefit of such a proactive tradeoff is illustrated with the motivational example given in Fig. 2 in Section III. We achieve distinctly longer extensions in sustained QoS for *Rock Player* and *Bodytrack* applications (i.e., 96% and 6.7×, respectively). In these two cases, our criticality-driven scheduling technique reduces the power on power-hungry big cores by exploiting the heterogeneity across the threads in terms of their criticality to overall QoS and reserving the big cores only for the threads that bring the most QoS gains. We detail our discussion on criticality awareness separately later in this section.

Fig. 11 illustrates the thermal profiles of applications with MAESTRO and default Android management. MAESTRO

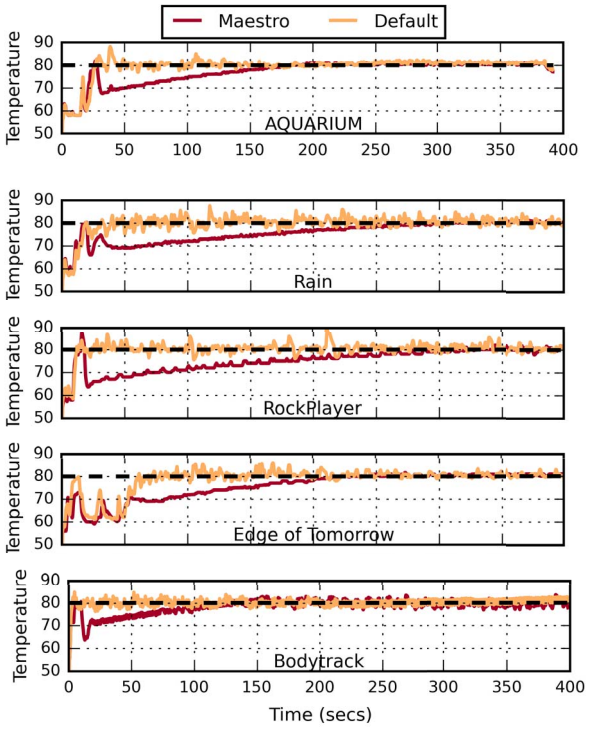

Fig. 11. Thermal profiles under MAESTRO and default Android management.

achieves lower temperature during the sustained duration. This extra thermal headroom (as high as 15 °C) allows MAESTRO to sustain the target QoS before thermal throttling starts to degrade performance. Such extra thermal headroom is achieved via proactive QoS tradeoff mechanism of MAESTRO (Section IV-B) as well as through the thermally efficient QoS control provided by QScale (Section IV-C).

C. Adaptive Runtime Behavior of MAESTRO

MAESTRO monitors the executing applications to detect throttling-prone continuous computations and can seamlessly adapt to changing workload patterns during runtime. Such changes can occur during typical daily usage scenarios. This section evaluates MAESTRO's ability to adapt to both inter- and intra-application changes in the computation patterns. To show interapplication adaptation, we design an experiment where a user session consists of both throughput-oriented CPU intensive computations as well as durations dominated by bursty computations.

Fig. 12 illustrates the runtime behavior of MAESTRO during a session where the user first launches *Aquarium* animation on the Web browser, followed by bursty image filtering operations (*Caman*) and concluded by opening the *Rain* animation. *Aquarium* and *Rain* correspond to applications with throughput-oriented and throttling-susceptible phases where we expect MAESTRO to activate QScale for enabling sustained performance. Based on the statistical properties estimated over the recent history of power profile, MAESTRO correctly identifies the continuous CPU-intensive computation phase in *Aquarium* and activates *QScale* with the normalized target QoS level of 0.7 after 30 s. After 70 s, as the user closes the *Aquarium* and launches *Caman* to perform various image editings online, MAESTRO switches off QScale and hands the

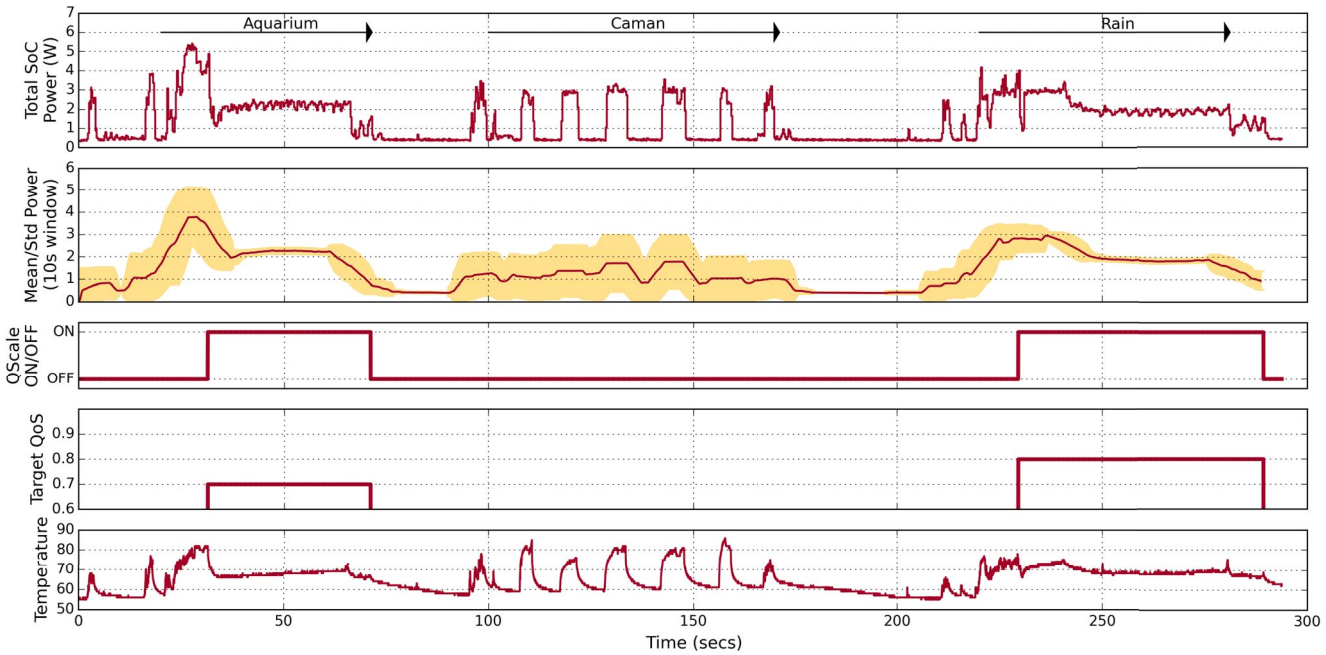

Fig. 12. Adaptive runtime behavior of MAESTRO. The user session consists of two throughput-oriented applications with heavy continuous workloads (i.e., *Aquarium* and *Rain*) interleaved by various UI-triggered bursty computations (app launches and image filtering operation in *Caman.js*). MAESTRO can successfully distinguish the continuous heavy computations in *Aquarium* and *Rain* that are prone to large throttling-induced QoS loss, and selectively activate QScale. Lower target QoS (i.e., 70% of the max) is selected for the *Aquarium* due to its high power profile with the goal of enabling a larger duration of sustained QoS. Bursty computations have distinguishably larger deviation (yellow area on second plot) within the power sampling window of 10 s.

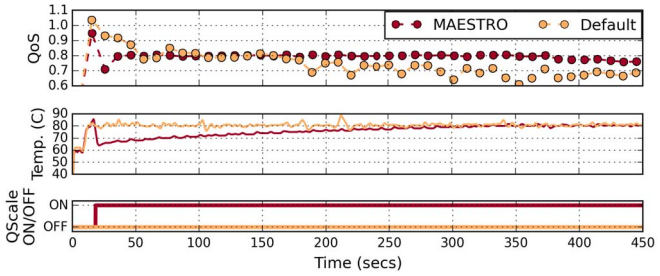

Fig. 13. Runtime behavior of MAESTRO for the RockPlayer video application. MAESTRO detects the heavy continuous computation once the video starts after the initial application launch and the user's menu traversals for video selection. Due to reduced CPU load on big cores with criticality-aware assignment of threads, the QoS degradation is substantially more gradual after the throttling starts when using MAESTRO.

control over to default Android management. The bursty nature of the computation with idle periods in between the events is manifested as the large deviation (yellow bars in second plot in Fig. 12) in the recent history of power samples. Finally, as the user launches another throttling-susceptible CPU intensive workload (i.e., *Rain*), MAESTRO once again activates QScale but with the 0.8 target QoS level.

Fig. 13 provides a detailed look into the *RockPlayer* video player application's runtime profile under MAESTRO and default Android management. After the application launch and several UI interactions across the application's menu options, the video starts to play and creates a continuously high computation pattern. MAESTRO detects such pattern and activates QScale after 20 s to stabilize the QoS around a 80% target QoS level. Temperature reduces abruptly and throttling does not occur until around 320 s, providing almost double the sustained duration for 80% QoS compared to default Android management. As we explain in detail in the next part, criticality-aware utilization of power-hungry cores brings substantial power reduction on the big cluster for this application, further enabling longer sustained durations.

D. Criticality-Aware Scheduling Versus HMP

In this section, we verify that the few critical threads identified per application (Section IV-C) determine the overall QoS and evaluate the power savings achievable via the criticality-aware thread assignment strategy within QScale where the power-hungry cores are restricted for critical threads of an application. Fig. 14(a) shows the power and average QoS for the throughput oriented applications when running with the baseline HMP scheduler and criticality-aware assignment of threads across big/LITTLE clusters. The DVFS policy in both settings is the Interactive governor. To avoid interference from thermal throttling, we set the fan to operate at the highest speed for these set of experiments. Criticality-aware assignment reduces the overall power by 20% for the *Rock Player* application by restricting the noncritical threads from operating on power-hungry big cores. For the case of *bodytrack* application, criticality-aware assignment of threads achieves 10% higher QoS than HMP. Thus, QScale is able to achieve the same QoS with HMP at a lower power by reducing the frequency. For *bodytrack*, HMP scheduler utilizes the big cluster for the worker threads that show high CPU load while leaving the initial main thread on the low-performance LITTLE core, limiting the performance gain. MAESTRO achieves higher QoS at a marginal power cost by, along with four other worker threads, moving this critical thread onto big cluster as well. We observe that baseline HMP and criticality-aware scheduling

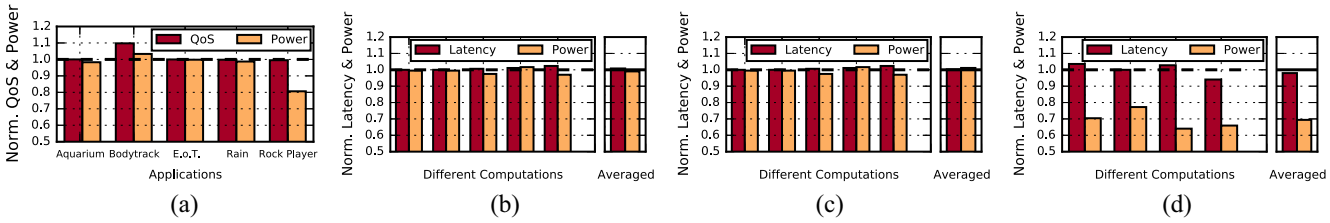

Fig. 14. QoS, latency, and power consumption achieved using criticality-aware scheduling. Data is normalized to HMP scheduling case. (a) QoS and average power consumption for various throughput-oriented applications. (b) Latency and average power consumption for various computational activities within *Caman.js*. (c) Latency and average power consumption for various computational activities within *Adobe PDF Reader*. (d) Latency/power for various computational activities within *Google Maps*.

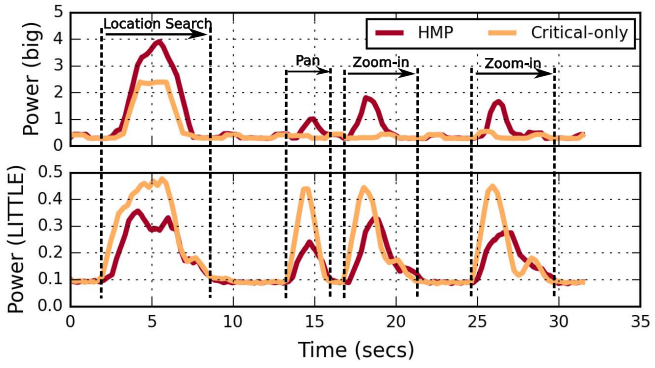

Fig. 15. CPU power profiles for various computational activities of Google Maps application showing the similar computational latencies at lower power with criticality-aware thread mapping.

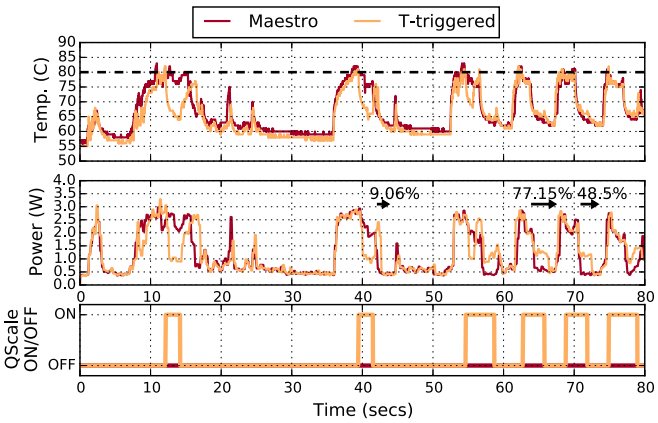

Fig. 16. Runtime behavior of *Adobe PDF reader* application while operating under MAESTRO and temperature-triggered sustained performance control policy. Temperature-triggered policy activates QScale when a critical thermal threshold is hit, and reverts back to default Android management when temperature is below 60 °C.

achieve similar power and QoS for the other throughput-oriented applications. Both policies perform the same actions as QoS is dictated by a few threads that also exhibit distinctly high CPU utilization.

Similar to Fig. 14(a), Fig. 14(b)–(d) shows the power and latency for different computational activities within *Caman*, *Pdf*, and *Maps* applications (e.g., swipe, zoom, search, etc.), respectively. Criticality-aware scheduling achieves similar latency with the HMP scheduler for all applications while also achieving the similar power consumption for *Caman* and *Pdf* cases. For the *Google Maps* application, 30% lower power consumption is achieved using the criticality-aware assignment of threads as averaged across all computations. Fig. 15 plots the power profiles of big and LITTLE CPU clusters for the *Google Maps* application and demonstrates the reduced power on the power-hungry big cores without incurring increased latencies. Start and ending of the computational activities are captured by detecting the transitions from idle power level as annotated in the figure.

E. Drawbacks of Temperature-Triggered QScale Activation

One naive approach to dynamically controlling application QoS for sustained performance would be to switch to a lower QoS level when the critical temperature threshold is reached. This section evaluates the limitations and drawbacks of relying on such an approach for detecting the throttling-susceptible QoS degradations (i.e., first block in Fig. 4). We consider a policy that activates QScale when the maximum SoC temperature limit (80 °C) is reached and switches back to default Android management when the temperature falls below 60 °C. Such a history-unaware and reactive temperature-triggered policy would suffer from various limitations. First, as the actions will be delayed until temperature limit is reached, system will heat up prematurely, which could have been avoided by tracking the high continuous computation patterns as we perform with MAESTRO. Second, volatile thermal violations can occur during bursty computations for short durations. Such volatile thermal peaks do not lead to large QoS loss as in continuous computation cases. A purely reactive temperature-triggered policy would incur frequent false alarms and cause premature switching to sustainable performance settings (i.e., activating QScale). We illustrate such a case for the *Adobe PDF Reader* application in Fig. 16. While MAESTRO can detect the bursty compute behavior and allows to maximize QoS with the default Android management, temperature-triggered (i.e., T-triggered) policy switches QScale on (bottom plot) at various points during runtime where temperature exceeds the maximum threshold (top plot). As annotated by arrows on the power profile (middle plot), falsely triggered switches to lower QoS settings leads to undesirable delays in the computation, impairing QoS by means of increased latency.

F. Android's Sustained Performance Mode

As an example sustained performance management scheme in real world, we study Android's sustained performance mode and demonstrate practical limitations due to its lack of application and QoS consideration. Our examination on Android's sustained performance API on a reference device (i.e., Nexus 6) indicates that the applications that request this mode of operation are forced to execute on thermally safe

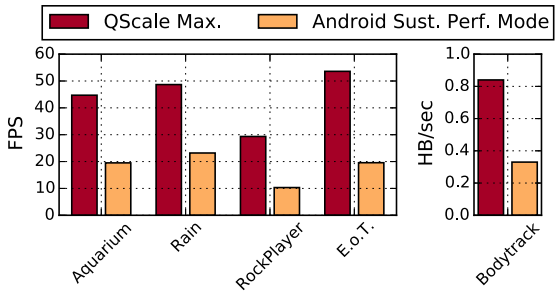

Fig. 17. Comparison of the maximum attainable QoS for our throughput-oriented applications to QoS levels obtained when operating under Android's sustained performance configuration. Selecting lower power operating modes for CPU and GPU to improve sustained performance with Android sustained performance mode configuration, without any QoS consideration, leads to substantially low QoS for such applications with high computation demand.

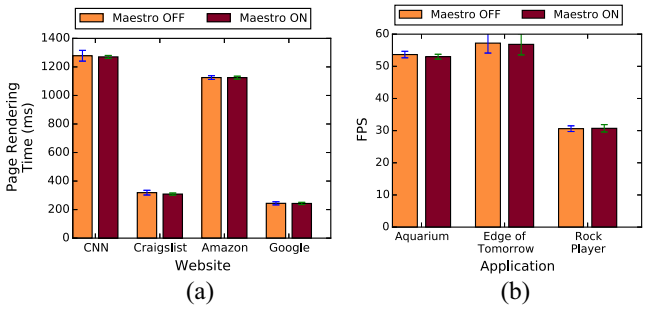

Fig. 18. Real system measurements to identify whether MAESTRO policy introduces an overhead that can cause performance degradation on (a) latency-sensitive and (b) throughput-oriented applications.

LITTLE cores with the maximum GPU frequency reduced to a medium level. While sustained durations can be extended with the reduced CPU and GPU power, such application and QoS oblivious scheme can provide drastically low QoS on applications where LITTLE cores are unable to provide the necessary computational capability. We illustrate this case in Fig. 17. We measure the maximum QoS attainable by QScale and by the reference implementation of Android's sustained performance configuration. The QoS levels in Android's sustained performance mode are at least 50% lower than the maximum QoS achievable for an application. For the frame-based applications, frame-rate (FPS) is consistently lower than 30 frames/s, reaching as low as 10 frames/s, which would likely deem application unusable from a user experience perspective. Thus, we argue the necessity of QoS consideration for sustained performance management policies.

G. Overhead Evaluation

In this part, we evaluate any performance overhead that could have been caused by MAESTRO's continuous operation in the background. We select applications from diverse sources to experiment using applications with varying CPU demand and parallelism requirements. We incorporate a set of CPU-intensive and throughput-oriented applications as well as selected websites with diverse computational needs from the BBench [22] browsing benchmark suite. We use the BBench suite as it provides precise timing of the webpage loading latency, which is the main performance metric. As illustrated in Fig. 18, the performance of the applications are not effected by the presence of MAESTRO. Overall CPU utilization of a single LITTLE core, which runs MAESTRO, is only 2.92% (0.82% in `usr` and 2.10% in `sys` mode). Our circular buffer implementation for sliding window also provides good locality of reference to minimize the energy and performance cost of memory accesses. Two core routines of MAESTRO which update the sliding window and estimate the value of the activation function based on mean/deviation (Section IV-A) take 1.86 and 476.7 μs (<0.05% of the 1 s invocation period), respectively. We measure execution time and CPU utilization at the lowest 1.0 GHz frequency of a low-performance LITTLE core to illustrate that even the worst case execution overhead of MAESTRO is minimal. The overall storage overhead of 2000 power samples in our 10 s sliding window would be 2000*4 bytes (float C type) which occupies only 0.3% of the cache space (2 MB) in our platform.

VI. RELATED WORK

A. Thermally Aware Runtime Management

Elevated power densities in mobile SoCs have led to an increased effort toward thermal management of mobile devices in particular. Prakash *et al.* [41] performed coordinated CPU-GPU DVFS to balance the sharing of thermal headroom between these two entities and maximize FPS. Similarly, ARM's *Intelligent Power Allocation* [50] shifts power across CPU and GPU based on expected performance return to improve performance under a temperature constraint. For conventional computer systems, control-theoretic [49] and predictive thermal management [48] techniques are widely studied to improve performance and energy efficiency while achieving smooth temperature control. MAESTRO builds upon a fundamentally different approach where, as opposed to greedily maximizing performance under thermal limits, we treat the thermal headroom as a resource that we seek to utilize for maximum durations while maintaining *just enough* QoS.

For multi/many core systems, various work present thermally efficient spatial allocation of threads in homogeneous [31], [46] and heterogeneous [32] systems. In fact, the intuition behind thermal-coupling aware mapping in our QScale policy [43] is similar to prior work. However, QScale [43] demonstrates, for the first time, the opportunities for thermally efficient core allocation on a mobile SoC by considering the application-specific CPU-GPU thermal couplings.

B. Application-Specific Optimizations for Mobile Applications

MAESTRO takes the computation behavior of an application into account to manage thermal headroom and QoS. Several prior studies also consider similar application characteristics to tailor energy optimization policies. Hashemi *et al.* [24] investigated the bursty compute behavior in Web applications for dynamic power management and derive a heuristic that sets the number of active cores based on per-thread instruction counts. For throughput-oriented applications such as gaming and video conversion, Rao *et al.* [42] profiled each application offline to determine the performance sensitivity of an application to CPU and memory DVFS and performs an application-specific control to minimize energy

under a performance target. Zhu *et al.* [53] proposed a framework to distinguish bursty and throughput-oriented events in a Web browser and allows to manage QoS and energy tradeoffs accordingly. While our focus is to, in addition to detecting the bursty versus throughput-oriented nature of computations, manage QoS-temperature tradeoff autonomously for sustainable QoS, the framework by Zhu *et al.* [53] could be complementary when crafting MAESTRO specifically for Web applications.

C. QoS Metrics and User Experience

Accurately defining QoS metrics is a challenging and open-ended task in computer systems research. In multicore memory system studies, QoS is manifested in terms of fairness [16]. Tail latency has been used as a measure of QoS for datacenter tasks [40]. For mobile systems, latency of user-triggered computations (e.g., application launch, GUI inputs, etc.) [17] and FPS in throughput-oriented applications (e.g., gaming, streaming) are found to be relevant proxies for user perceived QoS [13]. In this paper, we build upon the insights from prior work [13], [17] to determine relevant QoS metrics and requirements but provide a thermally aware way to dynamically manage QoS.

D. QoS-Centric Runtime Management

Various prior work has studied managing accuracy or QoS tradeoffs with power and energy considerations. PowerDial [28] automatically extracts application parameters to perform dynamic accuracy tradeoffs under a power cap. JouleGuard [26] provides a bandit-based learning solution to tune accuracy for meeting energy budgets. Such techniques rely on source-level instrumentation and approximate nature of specific applications (e.g., video encoding), which makes them hard to generalize to off-the-shelf mobile applications. Muthukaruppan *et al.* [36] proposed a budgeting framework on a big.LITTLE system where QoS levels of multiple single-threaded self-adaptive applications [27] are adjusted reactively upon power budget violations.

Majority of prior work on QoS-aware energy and thermal optimization for mobile applications target systems with *homogeneous CPUs*. Kadjo *et al.* [30] proposed a queueing model of CPU-GPU interactions in mobile games and design a CPU-GPU DVFS policy to meet maximum FPS targets while minimizing energy. Das *et al.* [15] controlled number of cores and DVFS levels simultaneously to optimize peak temperature and thermal cycling. Despite considering QoS targets in the learning approach, long convergence delays (i.e., minutes) pose practical limitations. Sahin *et al.* [44] proposed a thermally aware closed-loop DVFS scheduling policy to increase durations of sustained QoS. To guide a reinforcement learning-based thermal optimization policy, another work [14] proposes an online workload change detection technique. Their detection technique, however, does not identify workload patterns that are prone to thermally induced performance loss to make proactive decisions as we perform with MAESTRO.

For systems with heterogeneous CPUs, some other work target specific domains of mobile applications and consider QoS targets. Zhu and Reddi [54] exploited webpage characteristics to drive scheduling and DVFS decisions on a big.LITTLE system. Pathania *et al.* [38] used offline generated heuristics to guide big/LITTLE core allocation for multithreaded mobile games and achieved energy savings without any impact on maximum FPS. For server systems, Octopus-Man [40] reactively moves latency-critical Web applications to big or little CPUs while meeting a QoS constraint. Our prior QScale work [43], which is the basis of this paper, enables thermally efficient QoS tradeoffs by: 1) mapping threads on a big.LITTLE system by considering the QoS-criticality as well as the application-specific CPU-GPU thermal interactions and 2) performing closed-loop DVFS to precisely track QoS targets. However, QScale is agnostic to application-specific computation patterns and relies on users/programmers to specify QoS goals.

Different from these QoS-aware power/thermal optimization techniques, MAESTRO manages QoS *proactively* and *autonomously* in an *application-aware* manner. This allows MAESTRO to hit the sweet spot for autonomous QoS management of mobile applications by reclaiming sustainable performance in throttling-susceptible continuous computations (e.g., games) without sacrificing the latency in bursty applications.

E. Scheduling and Thermal Management in Heterogeneous CPUs

Some prior work aim at improving overall throughput in multiprogram workloads by guiding scheduling across heterogeneous CPUs based on hardware performance counters [33] and performance profiling on different cores [34]. For systems with cluster-level DVFS such as our platform, Pagani *et al.* [37] presented an efficient energy-optimization algorithm to compute V/F and core assignments for a set of independently running tasks with performance constraints. For mobile platforms, Hsiu *et al.* [29] prioritized foreground applications for execution on big cores over background applications. In this paper, we guide scheduling decisions based on the impact of individual threads on QoS based upon our novel observation on the heterogeneity of threads within an application. Prior work exploits varying criticality of threads to guide DVFS on homogeneous multicore systems (e.g., [11]).

Some other work targets thermal balancing through job allocation [47] and propose predictive thermal management methods [10], [48] for heterogeneous multicore CPUs. Other work [32] allocates available jobs across a set of heterogeneous processing elements to maximize throughput using a novel power density constraint. This paper focuses on effectively utilizing the available thermal headroom for maximum durations. Similar to ours, Paul *et al.* [39] have considered the CPU-GPU thermal coupling effects in modern SoCs in a runtime policy. Their work uses this insight to limit CPU frequency and allow for higher GPU performance while we present a thermally efficient and coupling aware thread mapping strategy.

This paper distinguishes from prior work in the following key aspects.

1) We show that greedily maximizing performance under thermal constraints can adversely impact user experience due to increased throttling over time. Thus, we aim at providing thermally efficient QoS tradeoffs to achieve sustained performance.
2) We demonstrate the profound impact of computation characteristics (i.e., bursty or throughput-oriented) on the throttling-induced QoS loss and QoS-temperature tradeoff decisions.

3) We design a thermal management technique that is application-specific. Our runtime management solution distinguishes long and continuous computations that are susceptible to throttling-induced QoS loss, and proactively manages QoS tradeoffs to utilize the thermal headroom for longer durations.
4) We observe that QoS is dominated by a few number of threads, which we exploit for scheduling on big.LITTLE to reduce the load on power-hungry cores.
5) Our policy dynamically guides core allocation decisions based on application-specific CPU-GPU thermal couplings.
6) We perform all experiments on real-life systems.

VII. Conclusion

Mobile applications exhibit fundamentally distinct computation patters across throughput-oriented workloads (e.g., video and gaming) and UI-driven bursty applications. These diverse categories of workloads not only have different QoS requirements but also exhibit largely disparate thermal profiles. Greedily improving performance brings diminishing returns in QoS due to throttling during continuous computations, making it challenging to achieve a sustainable QoS level. This paper proposed MAESTRO to identify application's compute behavior in runtime and manage QoS accordingly. For applications dominated by bursty tasks, MAESTRO allows to maximize QoS (i.e., to reduce latency) as the durations of thermal throttling is relatively short (i.e., less than a few seconds) and the idleness between the bursts allows to reduce temperature. On the other hand, MAESTRO proactively trades off QoS for the cases of continuous throughput-oriented computations with high power to enable sustained QoS over the use. We evaluate MAESTRO using real-life Android applications on a heterogeneous multicore platform. We demonstrate MAESTRO's ability to adapt to application behavior and autonomously manage QoS to improve durations of sustained performance by 41% to 6.7×.

We also see several exciting future directions for MAESTRO. One can alleviate the parameter tuning process by exploiting repetitive application runs [35] to gradually learn parameters online and by using recursive computations (e.g., exponential smoothing) to track statistical features. The nonlinear relation between FPS and frequency [21] can also be exploited to provide a more power-efficient QoS tradeoff. We believe exploring such further application-specific QoS tradeoffs (e.g., considering user requirements and frequency sensitivity of QoS and power) and evaluating the additional benefits on real users, to be a significant future research objective.

References

[1] *Aquarium*. Accessed: Mar. 21, 2018. [Online]. Available: http://webglsamples.org/aquarium/aquarium.html

[2] *Big.LITTLE Technology*. Accessed: Mar. 21, 2018. [Online]. Available: https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

[3] *CamanJS Image Editor*. Accessed: Mar. 21, 2018. [Online]. Available: http://camanjs.com/

[4] *Heterogeneous Multi-Processing*. Accessed: Mar. 21, 2018. [Online]. Available: https://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf

[5] *Linux CPUFreq*. Accessed: Mar. 21, 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[6] *Performance Management*. Accessed: Mar. 21, 2018. [Online]. Available: https://source.android.com/devices/tech/power/performance

[7] *Rain*. Accessed: Mar. 21, 2018. [Online]. Available: https://codepen.io/Sheepeuh/pen/cFazd

[8] *Revisiting SHIELD Tablet: Gaming Battery Life and Temperatures*. Accessed: Mar. 21, 2018. [Online]. Available: http://www.anandtech.com/show/8329/revisiting-shield-tablet-gaming-ux-and-battery-life

[9] *When Benchmarks Aren't Enough: CPU performance in the Nexus 5*. Accessed: Mar. 21, 2018. [Online]. Available: http://arstechnica.com/gadgets/2013/11/when-benchmarks-arent-enough-cpu-performance-in-the-nexus-5/

[10] G. Bhat, G. Singla, A. K. Unver, and U. Y. Ogras, "Algorithmic optimization of thermal and power management for heterogeneous mobile platforms," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 3, pp. 544–557, Mar. 2018.

[11] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2009, pp. 290–301.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2008, pp. 72–81.

[13] M. Claypool, K. Claypool, and F. Damaa, "The effects of frame rate and resolution on users playing first person shooter games," in *Proc. ACM/SPIE Multimedia Comput. Netw. (MMCN)*, 2006.

[14] A. Das, G. V. Merrett, M. Tribastone, and B. M. Al-Hashimi, "Workload change point detection for runtime thermal management of embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 8, pp. 1358–1371, Aug. 2016.

[15] A. Das *et al.*, "Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2014, pp. 1–6.

[16] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *Proc. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2010, pp. 335–346.

[17] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer, "Using latency to evaluate interactive system performance," *SIGOPS Oper. Syst. Rev.*, vol. 30, pp. 185–199, Oct. 1996.

[18] H. Falaki *et al.*, "Diversity in smartphone usage," in *Proc. 8th Int. Conf. Mobile Syst. Appl. Services (MobiSys)*, San Francisco, CA, USA, 2010, pp. 179–194.

[19] C. Gao *et al.*, "A study of mobile device utilization," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Philadelphia, PA, USA, 2015, pp. 225–234.

[20] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and touch-sensitive record and replay for android," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 72–81.

[21] U. Gupta *et al.*, "Adaptive performance prediction for integrated GPUs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, Nov. 2016, pp. 1–8.

[22] A. Gutierrez *et al.*, "Full-system analysis and characterization of interactive smartphone applications," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Nov. 2011, pp. 81–90.

[23] M. Halpern, Y. Zhu, and V. Reddi, "Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction," in *Proc. IEEE 22th Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2016, pp. 64–76.

[24] M. Hashemi, D. Marr, D. Carmean, and Y. N. Patt, "Efficient execution of bursty applications," *IEEE Comput. Archit. Lett.*, vol. 15, no. 2, pp. 85–88, Jul./Dec. 2016.

[25] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. New York, NY, USA: Wiley, 2004.

[26] H. Hoffmann, "JouleGuard: Energy guarantees for approximate applications," in *Proc. 25th Symp. Oper. Syst. Principles (SOSP)*, Monterey, CA, USA, 2015, pp. 198–214.

[27] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments," in *Proc. 7th Int. Conf. Auton. Comput. (ICAC)*, Washington, DC, USA, 2010, pp. 79–88.

[28] H. Hoffmann *et al.*, "Dynamic knobs for responsive power-aware computing," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2011, pp. 199–212.

[29] P.-C. Hsiu, P.-H. Tseng, W.-M. Chen, C.-C. Pan, and T.-W. Kuo, "User-centric scheduling and governing on mobile devices with big.LITTLE processors," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 1, p. 17, 2016.

[30] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz, "A control-theoretic approach for energy efficient CPU-GPU subsystem in mobile platforms," in *Proc. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2015, pp. 1–6.
[31] H. Khdr, S. Pagani, M. Shafique, and J. Henkel, "Thermal constrained resource management for mixed ILP-TLP workloads in dark silicon chips," in *Proc. ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2015, pp. 1–6.
[32] H. Khdr *et al.*, "Power density-aware resource management for heterogeneous tiled multicores," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 488–501, Mar. 2017.
[33] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, 2010, pp. 125–138.
[34] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proc. 31st Annu. Int. Symp. Comput. Archit. (ISCA)*, 2004, pp. 64–75.
[35] X. Li, G. Chen, and W. Wen, "Energy-efficient execution for repetitive app usages on big.LITTLE architectures," in *Proc. 54th Annu. Design Autom. Conf. (DAC)*, 2017, p. 44.
[36] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Proc. Design Autom. Conf. (DAC)*, Austin, TX, USA, 2013, pp. 1–9.
[37] S. Pagani, A. Pathania, M. Shafique, J.-J. Chen, and J. Henkel, "Energy efficiency for clustered heterogeneous multicores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 5, pp. 1315–1330, May 2017.
[38] A. Pathania, S. Pagani, M. Shafique, and J. Henkel, "Power management for mobile games on asymmetric multi-cores," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, Jul. 2015, pp. 243–248.
[39] I. Paul, S. Manne, M. Arora, W. L. Bircher, and S. Yalamanchili, "Cooperative boosting: Needy versus greedy power management," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2013, pp. 285–296.
[40] V. Petrucci *et al.*, "Octopus-Man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Burlingame, CA, USA, Feb. 2015, pp. 246–258.
[41] A. Prakash, H. Amrouch, M. Shafique, T. Mitra, and J. Henkel, "Improving mobile gaming performance through cooperative CPU-GPU thermal management," in *Proc. Annu. Design Autom. Conf. (DAC)*, Austin, TX, USA, 2016, pp. 1–6.
[42] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and Y. Handong, "Application-specific performance-aware energy optimization on android mobile devices," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Austin, TX, USA, 2017, pp. 169–180.
[43] O. Sahin and A. K. Coskun, "QScale: Thermally-efficient QoS management on heterogeneous mobile platforms," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, 2016, p. 125.
[44] O. Sahin, P. T. Varghese, and A. K. Coskun, "Just enough is more: Achieving sustainable performance in mobile devices under thermal limitations," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, 2015, pp. 839–846.
[45] W. Seo, D. Im, J. Choi, and J. Huh, "Big or little: A study of mobile interactive applications on an asymmetric multi-core platform," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Atlanta, GA, USA, 2015, pp. 1–11.
[46] M. Shafique, D. Gnad, S. Garg, and J. Henkel, "Variability-aware dark silicon management in on-chip many-core systems," in *Proc. Design Autom. Test Europe (DATE)*, 2015, pp. 387–392.
[47] S. Sharifi, A. Coskun, and T. S. Rosing, "Hybrid dynamic energy and thermal management in heterogeneous embedded multiprocessor SoCs," in *Proc. Asia South Pac. Design Autom. Conf. (ASP-DAC)*, 2010, pp. 873–878.
[48] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras, "Predictive dynamic thermal and power management for heterogeneous mobile platforms," in *Proc. Design Autom. Test Europe (DATE)*, 2015, pp. 960–965.
[49] K. Skadron, T. Abdelzaher, and M. R. Stan, "Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2002, p. 17.
[50] X. Wang. *Intelligent Power Allocation*. Accessed: Mar. 21, 2018. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dto0052a/DTO0052A_intelligent_power_allocation_white_paper.pdf
[51] K. Yan, X. Zhang, J. Tan, and X. Fu, "Redefining QoS and customizing the power management policy to satisfy individual mobile users," in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2016, pp. 1–12.
[52] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda, "Panappticon: Event-based tracing to measure mobile application and platform performance," in *Proc. Int. Conf. Hardw. Softw. Codesign Syst. Synth. (CODES+ISSS)*, Montreal, QC, Canada, Sep. 2013, pp. 1–10.
[53] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-based scheduling for energy-efficient QoS (eQoS) in mobile Web applications," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 137–149.
[54] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile Web browsing on big/little systems," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2013, pp. 13–24.

Onur Sahin received the B.Sc. degrees in electronics engineering and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 2012 and 2013, respectively. He is currently pursuing the Ph.D. degree in computer engineering with Boston University, Boston, MA, USA.

His current research interests include system software, computer architecture, mobile computing, power/thermal efficiency, and security of mobile platforms.

Lothar Thiele (M'86) received the Diplomingenieur and Dr.-Ing. degrees in electrical engineering from the Technical University of Munich, Munich, Germany, in 1981 and 1985, respectively.

He joined the Information Systems Laboratory, Stanford University, Stanford, CA, USA, in 1987. In 1988, he was the Chair of Microelectronics with the Faculty of Engineering, University of Saarland, Saarbrücken, Germany. He joined ETH Zürich, Zürich, Switzerland, as a Full Professor of Computer Engineering, in 1994, where he has been the Associate Vice President of Digital Transformation since 2017. In 2004, he joined the German Academy of Sciences Leopoldina, Schweinfurt, Germany. His current research interests include models, methods, and software tools for the design of embedded systems, embedded software, and bioinspired optimization techniques.

Mr. Thiele was a recipient of the Dissertation Award of the Technical University of Munich, in 1986, the Outstanding Young Author Award of the IEEE Circuits and Systems Society in 1987, the Browder J. Thompson Memorial Award of the IEEE in 1988, the IBM Faculty Partnership Award in 2000 and 2001, the Honorary Blaise Pascal Chair of University of Leiden, in 2005, and the "EDAA Lifetime Achievement Award" in 2015. He is an Associate Editor of *ACM Transactions on Sensor Networks*, *ACM Transactions on Cyberphysical Systems*, the IEEE TRANSACTION ON INDUSTRIAL INFORMATICS, the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, the *Journal of Real-Time Systems*, the *Journal of Signal Processing Systems*, the *Journal of Systems Architecture*, and the *Integration, the VLSI Journal*. He joined the National Research Council of the Swiss National Science Foundation in 2013. Since 2009, he has been a member of the Foundation Board of Hasler Foundation. Since 2010, he has been a member of the Academia Europaea.

Ayse K. Coskun (M'06–SM'16) received the M.S. and Ph.D. degrees in computer science and engineering from the University of California at San Diego, San Diego, CA, USA.

She is an Associate Professor with the Electrical and Computer Engineering Department, Boston University (BU), Boston, MA, USA. She was with Sun Microsystems (currently, Oracle), San Diego, prior to her current position at BU. Her current research interests include energy and temperature awareness in computing systems, covering novel architectures such as 3-D-stacked systems, cloud and HPC data centers, and mobile/embedded systems.

Prof. Coskun was a recipient of the NSF CAREER Award and the IEEE CEDA Early Career Award. She currently serves as an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS.