QScale: Thermally-Efficient QoS Management on Heterogeneous Mobile Platforms

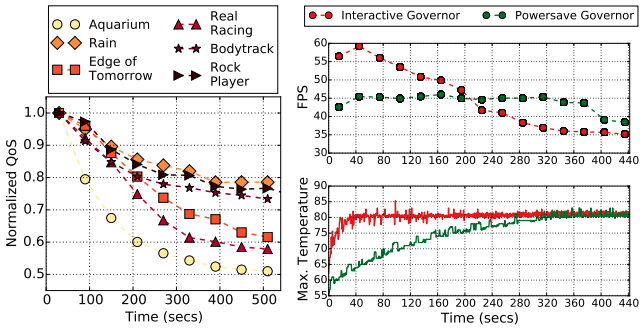
Onur Sahin and Ayse K. Coskun
ECE Department, Boston University, Boston, MA, USA
{sahin, acoskun}@bu.edu

ABSTRACT

Single-ISA heterogeneous mobile processors integrate low-power and power-hungry CPU cores together to combine energy efficiency with high performance. While running computationally demanding applications, current power management and scheduling techniques greedily maximize quality-of-service (QoS) within thermal constraints using power-hungry cores. We show that such an approach delivers short bursts of high QoS, but also causes severe QoS loss over time due to thermal throttling. To provide mobile users with *sustainable QoS* over extended durations, this paper proposes *QScale*. QScale is a novel *thermally-efficient QoS management* framework for mobile devices with heterogeneous multi-core CPUs. QScale leverages two novel observations to provide thermally-efficient QoS: (1) threads of a mobile application exhibit significant heterogeneity, which can be exploited during scheduling; (2) thermal efficiency of core allocation decisions is significantly altered by thermal interactions across system-on-a-chip (SoC) components and application characteristics. QScale coordinates closed-loop frequency control with thermally-efficient scheduling to deliver the desired QoS with minimal exhaustion of processor thermal headroom. Our experiments on a state-of-the-art heterogeneous mobile platform show that QScale meets target QoS levels while minimizing heating, achieving up to 8x *longer durations of sustainable QoS*.

1. INTRODUCTION

Single-ISA heterogeneous multi-core processors [22] (e.g., ARM big.LITTLE [3]) have been commonly adopted in recent mobile SoCs. Such designs offer large dynamic power and performance ranges, and achieve significant energy savings in mobile applications with widely varying performance demands [25, 28, 32]. Current heterogeneous CPU designs incorporate multiple aggressive power-hungry cores (i.e., out-of-order and speculative multi-issue pipelines [23]) to handle performance demanding tasks. Power consumption of modern mobile SoCs with such high-power big cores can reach well above the thermal design power (TDP) (e.g., 8W peak power in Samsung Exynos 7 despite a TDP of 3.5W [13]) and the maximum chip temperature can quickly elevate to critical levels [30]. The problem accelerates further due to inherent limitations in cooling capabilities of mobile devices. Built-in thermal throttling policies in smartphones mitigate thermal violations by slowing down the CPU (i.e., via dynamic voltage/frequency scaling (DVFS), power gating etc.), and thus, they sacrifice QoS experienced by the user [6, 27]. As a result, temperature plays a critical role in current mobile devices' capability to sustainably deliver satisfactory performance levels to the end-user.



(a) (b)

Figure 1: Performance degradation over time due to thermal throttling on a Odroid-XU3 platform (left) and a case for QoS-temperature tradeoff for longer sustainable QoS (right).

Motivational Example: Consider the case in Figure 1. The left plot demonstrates the throttling-induced QoS degradation over time for 6 applications running on a Odroid-XU3 development platform powered with a big.LITTLE processor when using the default DVFS and scheduling scheme. Throttling[1] incurs up to 22%-48% QoS degradation as the applications are continuously run over extended durations. Figure 1b shows the FPS[2] and the maximum chip temperature for an 8-minute run of *Real Racing* game, illustrating the benefits of *QoS-temperature tradeoff* for achieving longer sustainable QoS. The default Interactive governor [1] provides the highest QoS initially, but quickly exhausts the processor thermal headroom and exhibits significant QoS degradation over time due to thermal throttling. On the other hand, the Powersave governor [1], which always uses the lowest available frequency, *sustains the QoS above 45FPS for almost twice as long* since heating is slowed down and throttling is mitigated. The challenge here lies in determining the best scheduling and DVFS combination that maximizes the sustained QoS duration under dynamically varying levels of QoS requirements for a wide range of applications, and achieving this in face of both hardware and application-level (e.g., varying CPU demands of threads) heterogeneity. Addressing this challenge requires a QoS management strategy that minimizes heating without violating QoS requirements.

ICCAD '16, November 07-10, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2966986.2967066

[1] Thermal throttling is incurred via a reactive DVFS and thread migration policy with a 80°C maximum temperature limit as explained in Section 2.

[2] We use frames-per-second (FPS) as the measure of quality-of-service (QoS) experienced by the user.

This strategy should also be aware of the asymmetric heating behavior and inter-component thermal couplings in the SoC to make thermally-aware decisions and achieve minimum temperature.

This paper proposes *QScale*, a thermally-efficient QoS management framework for mobile platforms with heterogeneous CPUs. QScale uses both CPU DVFS and thread scheduling as control knobs. The goal behind QScale is to minimize heat generation while precisely delivering the desired QoS levels so as to provide a consistent user experience for the maximum durations. QScale leverages two key observations. (1) Application threads have significant heterogeneity in terms of their criticality to user-experience and such information can guide runtime DVFS and scheduling decisions to achieve lower temperatures. While recent work [11, 28] has studied multi-threading in mobile applications at a coarser granularity to point to the low inherent thread-level parallelism (TLP), leveraging user-experience critical threads in scheduling and DVFS decisions at runtime to optimize temperature is a novel aspect of our work. (2) Significant thermal interactions occur between the GPU and CPU cores and thermal efficiency of thread-to-core mappings widely changes depending on the CPU-GPU thermal coupling and application characteristics. Prior work [19, 25] has proposed integrated CPU-GPU DVFS policies for energy minimization but primarily focused on the performance interactions between CPU and GPU. QScale focuses on an orthogonal problem and monitors the thermal interactions between the CPU and GPU to identify thermally-efficient CPU cores for execution and, in this way, minimize heating.

In summary, we make the following specific contributions:

- To the best of our knowledge, our work is the first to identify the throttling-induced QoS degradation over extended durations in the state-of-the-art heterogeneous mobile CPUs. Using real-life experiments, we demonstrate that greedily exhausting the thermal headroom for boosting instant QoS, without considering the future impacts, can lead to significant QoS loss when applications run longer.
- We demonstrate that CPU-GPU thermal coupling is application dependent, and propose a mechanism for thermally-efficient, application-aware selection of big cores for execution (Section 3.1). We show that, combined with the thermally-efficient core allocation, identifying the relatively few number of *QoS-critical threads* (Section 3.2) in mobile applications that exhibit low TLP and scheduling those threads on thermally-efficient CPU cores substantially slows down heating.
- We propose *QScale*, which combines offline thermal coupling and thread criticality information with a runtime thermally-efficient scheduling and control-theoretic DVFS controller (Section 3.3). QScale is able to meet target QoS requirements for a set of real-life mobile applications. We implement QScale on a state-of-the-art heterogeneous mobile platform and demonstrate up to 8x longer durations of sustainable QoS (Section 4).

2. EXPERIMENTAL METHODOLOGY

This section presents our experimental testbed, data monitoring/collection methodology and application set.

Experimental Platform: All of our measurements and evaluations are based on real-life experiments on a contemporary mobile system. We use an Odroid-XU3 mobile development board that comprises of the Samsung Exynos 5422 SoC (which is included in the Samsung Galaxy S5 smartphone). The board runs Android 4.4 KitKat as the OS. The Exynos 5422 SoC implements a big.LITTLE heterogeneous CPU architecture [3] with quad-core big (A15) and little (A7) CPU clusters. The A15 is a high performance/power multi-issue out-of-order processor while A7 is a low performance/power core with simple 8-stage in-order pipeline [3]. The A15 core supports 9 frequency levels from 1.2 GHz to 2 GHz. The A7 core operates on 5 frequency levels between 1 GHz and 1.4 GHz. The frequency scaling decisions occur at a cluster-level as the cores within a cluster share the same voltage/frequency domain. The Exynos 5422 SoC also integrates Mali-T628 GPU, which supports 6 frequency levels ranging from 177 MHz to 543 MHz. A default mechanism scales the GPU frequency based on utilization.

Table 1: Summary of applications and QoS metrics.

Application	Category	QoS Metric
Bodytrack	Computer Vision	Heartbeats/sec
Real Racing	Gaming	FPS
Edge of Tomorrow	Gaming	FPS
Aquarium	Graphics/WebGL	FPS
Rain	Graphics/WebGL	FPS
Rock Player	Video Playback	FPS

Measurement Methodology: The board is equipped with a Texas Instruments INA231 power monitoring unit and allows measuring power consumptions of the A15 and A7 clusters, the GPU, and the memory individually. The platform provides temperature sensors for each of the 4 big cores as well as for the GPU. We measure FPS by querying the logs generated by the *SurfaceFlinger* Android system service.

Applications: Table 1 provides a list of applications that we use in our experiments. We run the bodytrack computer vision application from the PARSEC suite [8] where the frame-rate (or heartbeats/sec) is dynamically monitored by instrumenting the application with the Heartbeats framework [15]. Two gaming applications, *Edge of Tomorrow* and *Real Racing*, are chosen as representatives of modern gaming applications. We use *Rock Player* video player application to display a 1 minute HD video and loop the video to experiment with longer durations. We use a timing-based record/replay tool, RERAN [12], to automate the execution by injecting a pre-recorded set of GUI events. Rain [5] and Aquarium [2] are web-based online animations that we execute within the Chrome web browser. All of our applications are multi-threaded. Figure 2 shows the large number of software threads in mobile applications. Number of threads in bodytrack is configured to 8, which equals to the total number of CPU cores.

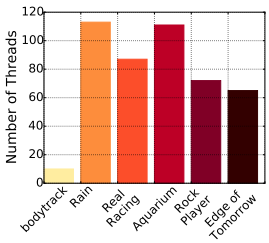


Figure 2: #Threads.

In addition, we write two custom microbenchmark applications for use during the offline thermal coupling characterization process. As a GPU microbenchmark, we write an OpenCL program that repeatedly offloads a matrix multiplication kernel to GPU. CPU portion of this microbenchmark is lightweight (<1.5% CPU utilization) and is always pinned to a low-power little core. CPU microbenchmark continuously performs floating-point multiplications.

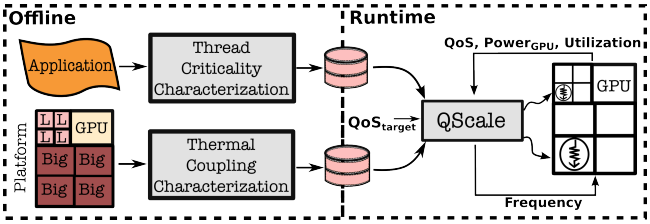


Figure 3: An overview of the proposed framework.

Runtime Management: Our platform uses an external fan for cooling. As fans are not available in commercial devices, we disable the fan control and implement a baseline reactive DVFS throttling policy. This policy reactively increments/decrements the maximum DVFS state of big cores every second if the maximum temperature is lower/higher than 80°C. By modifying the maximum DVFS level, the throttling policy forces the CPU DVFS governors to use lower frequencies without disabling their operation. If 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. Baseline CPU DVFS policy is the *Interactive governor* [1], which is default in most Android devices. This governor scales the CPU frequency to the maximum if the utilization is higher than a threshold. Once scaled to the highest, CPU frequency is not scaled down for at least 20ms to maximize responsiveness. The baseline HMP scheduler [4] determines thread-to-core mappings. HMP 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*. QScale operates every second and uses *cpufreq* and *sched_setaffinity* interfaces to control the frequency and the thread mappings for an application.

3. QSCALE DESIGN

This section describes QScale, a novel *thermally-efficient QoS* management framework for heterogeneous mobile platforms. Figure 3 presents a high level flow of our framework. During the offline phase, we use a set of CPU/GPU microbenchmarks to identify the thermal coupling between the big cores and the GPU, and derive lightweight heuristics for runtime thermally-efficient core allocation (Section 3.1). We also identify the threads of an application that are critical to user-experience during the offline phase (Section 3.2). Runtime management (Section 3.3) monitors the application's CPU and GPU usage and leverages the offline-generated heuristics to identify the most thermally-efficient big cores for executing the QoS-critical threads of an application. The runtime management policy also performs closed-loop DVFS control to precisely deliver the desired QoS.

3.1 Thermally-Efficient Core Allocation

GPU-CPU Thermal Coupling: First, we demonstrate the thermal coupling effect between the GPU and the big cores by running our GPU microbenchmark. In this experiment, the GPU operates at peak utilization and at the highest frequency for 1 minute. Figure 4 shows the resulting temperature increase (from an initially cold system) and the maximum

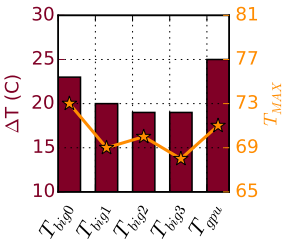
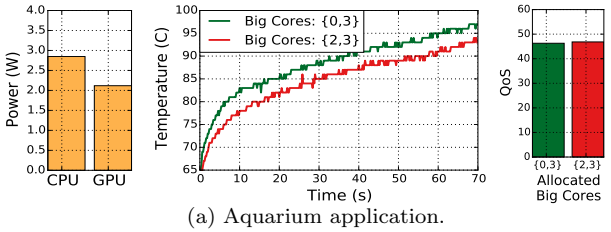
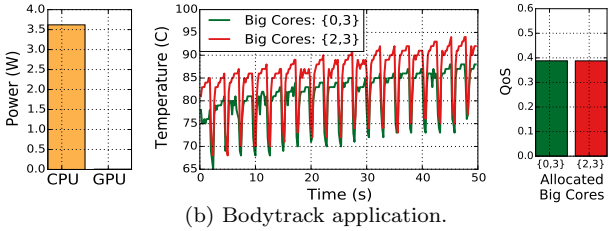


Figure 4: GPU thermal coupling in Exynos 5422.



(a) Aquarium application.



(b) Bodytrack application.

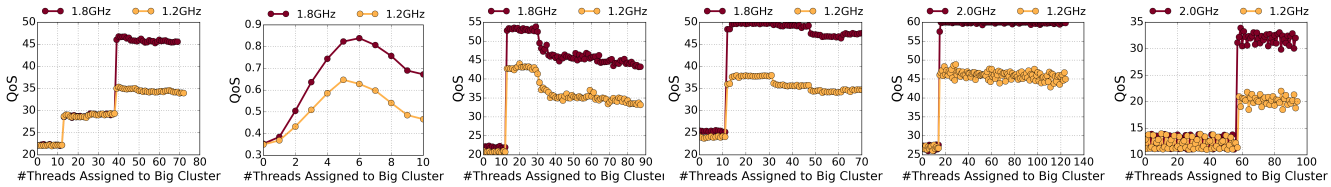
Figure 5: Power breakdown (left), temperature traces (middle) and QoS (right) for aquarium and bodytrack applications under different big core assignments ({0,3} and {2,3})

temperatures of the GPU and CPU cores. As the GPU temperature increases from 46°C to 71°C by 25°C, we measure significant heating on all idle big cores. *Core*0 suffers the most from thermal coupling and its temperature increases by 23°C. Cores heat up at different rates due to their different locations on chip and varying proximities to the GPU.

Need for Coupling-aware Core Allocation: Next, we demonstrate that the impact of GPU-CPU thermal coupling is *application-dependent*. Figure 5 shows the temperature profiles of two real-life applications with distinct CPU and GPU usage when the two highest utilization threads are pinned to two different set of big cores[3] ($\{0,3\}$ and $\{2,3\}$). We do not show the other allocation cases for clarity. Throttling is disabled to avoid interference with measurements. $\{0,3\}$ allocation results in the quickest increase in temperature among all possible allocation scenarios for *aquarium*, while the same allocation achieves the lowest temperature for *bodytrack*. $\{2,3\}$ results in the highest temperature for *bodytrack*, while achieving a lower temperature than $\{0,3\}$ for *aquarium*. We observe that *Core*0 provides thermally-efficient operation when the GPU is 'cool', but its temperature can quickly elevate otherwise. Note that both allocations achieve the same QoS. We propose an *offline characterization* step to capture this interplay between the thermal-efficiency of CPU cores and the GPU-CPU thermal coupling.

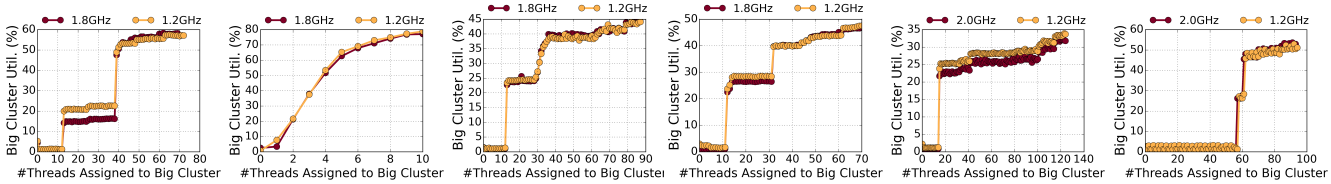
Offline Characterization: The aims of this offline step are to characterize how the GPU-CPU thermal coupling affects the thermal efficiency of each big core and to derive a strategy for thermally-efficient core allocation. In order to control CPU and GPU independently and obtain an accurate characterization, we use the microbenchmarks described in Section 2. We sweep the GPU power level up to $2W$ in $0.1W$ increments by tuning the input size of our GPU microbenchmark. For each GPU power level, we run the CPU microbenchmark on each big core for 1 minute and record the ordering of cores from the lowest to the highest peak temperature. We identify that this ordering changes at two GPU power levels ($Thr1 = 0.25W$ and $Thr2 = 1.2W$). We convert this characterization into a runtime heuristic shown in Table 2. For a given runtime GPU power level, the policy

[3] Core0 to Core3 correspond to cpu4 to cpu7 under the /sys/devices/system/cpu/ file path within the Linux file system on the Odroid-XU3 platform.



(a) Aquarium (b) Bodytrack (c) EoT (d) Rain (e) Racing (f) RP

Figure 6: QoS scaling achieved by moving individual application threads from the little to the big cluster.



(a) Aquarium (b) Bodytrack (c) EoT (d) Rain (e) Racing (f) RP

Figure 7: Increase in big cluster usage as individual application threads are moved from the little to the big cluster.

Table 2: Thermal coupling aware core allocation policy.

Condition	Order of Big Core Allocation
$Thr2 \leq P_{GPU}$	Core3, Core1, Core2, Core0
$Thr1 \leq P_{GPU} < Thr2$	Core3, Core1, Core0, Core2
$P_{GPU} < Thr1$	Core3, Core0, Core1, Core2

uses the core orderings from the characterization step to determine which CPU cores to allocate first. $Core3$ provides the lowest temperature across all GPU power levels and, thus, is always allocated first. Thermal efficiency of $Core0$, however, strongly depends on the GPU usage. Due to close proximity to GPU, this core results in the worst peak temperature when the GPU power is high. GPU acts as a heat spreader when its power is low and $Core0$ achieves the lowest temperature in this case. Between the two GPU power thresholds, thermal efficiency of $Core0$ falls between $Core1$ and $Core2$.This empirical methodology allows to identify the thermally-efficient big cores on a GPU-CPU thermally coupled system without requiring any packaging or floorplan details, which are not usually provided by vendors.

3.2 Thread Criticality in Mobile Applications

We propose to guide scheduling decisions on big.LITTLE by identifying the threads that are critical to user-experience, as opposed to leveraging the coarse-grained utilization metrics used in current schedulers [4]. Our novel observation behind this approach is that the overall QoS of mobile applications is dominated by a relatively few number of *QoS-critical threads* (compared to number of available cores). This observation is in line with the recent work [11, 28], which has identified that majority of mobile applications do not benefit from increased number of cores. We identify the QoS-critical threads of an application via a simple offline characterization process. Our approach is to prevent the big cores from quickly exhausting the thermal headroom by reserving them only for QoS-critical threads, which require higher performance. We use the low-power little cores at the highest frequency (1.4GHz) for other non-critical threads.

Offline Characterization: The aim of this offline characterization step is to identify the QoS-critical threads of an application that provide the highest QoS gains when allocated on a big core. We perform this characterization on each of our 6 applications. First, we allocate all the application threads to little cluster. We increment the number of application threads allocated to the big cluster by one thread at a time, and for each case, we record the average QoS and big cluster utilization. We run every scheduling configuration for 30 seconds. Figure 6 shows the QoS scaling and Figure 7 shows the increase in big cluster utilization recorded during this characterization step. This offline characterization reveals that QoS is dictated by a small number of critical threads for most applications. For instance, moving a single critical thread to the big cluster achieves close to peak QoS for the *Edge of Tomorrow*, *Real Racing*, *Rock Player* and *Rain* applications. A more balanced criticality is observed for *bodytrack* from the PARSEC suite [8], for which aggressively moving threads to big cluster for performance leads to QoS degradation. For *bodytrack*, despite its low CPU utilization, assigning the initial helper thread (first thread) to big cluster along with 4 worker threads significantly improves performance. To explore whether thread criticality changes upon different application inputs or at different frequencies, we perform the same characterization at high (1.8GHz) and low (1.2GHz) frequencies, run the gaming applications with different set of recorded GUI interactions and play *Rock Player* with different inputs (HD video files). Our results have shown that QoS is still dictated by the same critical threads. Overall, the number of critical threads per application are identified as 5 for *bodytrack*, 2 for *aquarium* and *rain* and, 1 for the others.

The output of the offline characterization process, which is communicated to the runtime management, is a set of `<Application, Thread Name, ThreadId Offset, Average CPU Utilization>` tuples. `ThreadId Offset` corresponds to the offset from the ID of the first thread launched by the parent process, and is used when thread names conflict. Similarly, in case the offsets change during application launch, we record the per-thread CPU usage as another proxy for uniquely identifying the QoS-critical threads.

While our approach requires offline thread profiling for every application, we argue that such approach is profitable specifically for mobile applications for various reasons: (1) Users will likely run the same application many times in the device life-cycle; (2) such offline profiling of applications can be automatically performed on a device without user interference (e.g., while device is left in charging) using Android record/replay tools [12, 18] as we have done in this work using RERAN [12].

$\mathbf{QoS_{target}}$: Target QoS level
Threads[**A**] : List of threads for application A
Max_State : Max. state allowed by the throttling policy.
At beginning:
$LittleCluster \leftarrow \{Threads[A]\}$
Every second:
Perform thread-to-core mappings (Algorithm 2)
$target\_state = PIController()$
Clip($target\_state$, Min_State, Max_State)
Actuator($target\_state$)

Algorithm 1: QScale's top-level runtime policy.

3.3 Runtime Control Policy

Overview: The goal of QScale's runtime policy is to deliver desired QoS levels while minimizing temperature by coordinating thermally-efficient scheduling with DVFS. It monitors the GPU power dissipation to select the most thermally-efficient set of big cores for executing the QoS-critical threads of an application. The policy also performs control-theoretic DVFS to precisely meet QoS targets. Target QoS can be dynamically adjusted upon user request, autonomously set by the system-level policies (e.g., based on battery level or thermal status), or statically fixed to bare minimum levels based on the limitations of user perception [10, 31].

Algorithm 1 provides an overview of our top-level runtime control algorithm. The policy initially assigns the threads to the little cluster and, at every second, invokes our mapping policy (Algorithm 2) to partition the application threads among the big and little clusters in a thermally-efficient manner. DVFS level for the next control interval is calculated by the closed-loop controller (Figure 8). The policy avoids thermal violations by clipping the controller output (e.g., next DVFS state) to a range below the maximum level, which is determined by the thermal throttling policy.

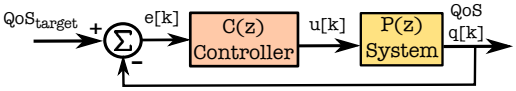


Figure 8: Feedback-based performance state control.

Algorithm 2 illustrates the thread-to-core mapping flow. If the number of critical threads is less than 4, there exists opportunity to lower temperature by making thermally-aware mapping. In this case, we sequentially bind the *critical thread with the highest CPU usage to the next most thermally-efficient core*. Otherwise, the policy allocates the critical threads to the big cluster and uses default Linux load balancer for task mappings within the cluster. The order of core allocations is determined based on the GPU power.

Performance States in QScale: QScale's feedback controller uses DVFS on big cores as a control knob. In addition to DVFS, to bridge the performance gap between little and big cores, we implement 4 migration-based (M) states (Figure 9). We collectively refer to DVFS and M-states as performance states. Migration states perform frequent (every 100ms) switching between the $\texttt{Little}_{1.4GHz}$ and $\texttt{Big}_{1.2GHz}$ operation at 20%, 40%, 60%, 80% duty-cycles within the control interval. The number of switchings is maximized to provide the minimum possible duration of continuous little core operation. This minimizes user

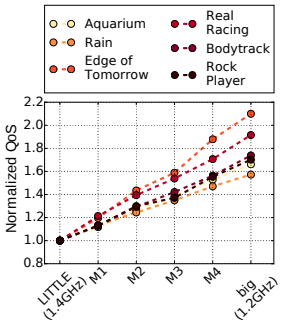


Figure 9: M-States.

$\mathbf{CriticalThreads}$[**A**] : QoS critical threads for application A
$\mathbf{P_{GPU}}$: Current GPU Power
$N = length(CriticalThreads[A])$
if $N >= 4$ **then**
  Assign $\{CriticalThreads[A]\}$ to big cluster.
  Use default Linux task mapping.
  return
else
  Sort $CriticalThreads[A]$ in descending order of utilization.
  for $i = 1$ ***to*** N **do**
    $TargetCore$ = CoreAllocation(P_{GPU}, i) (Table 2)
    $TargetCore \leftarrow CriticalThreads[A][i]$
  end
end

Algorithm 2: Thread-to-core mapping policy.

perceived latency. We do not migrate more frequently than every 100ms to avoid migration overhead [20, 24]. On the little cluster, only the highest DVFS operation (1.4GHz) is considered as it provides thermally-safe operation and no lower states are needed to improve QoS sustainability.

Closed-loop controller design: The closed-loop controller estimates the performance state for the next interval that will meet the target QoS. The error term ($e[k]$) (Figure 8) simply corresponds to the current offset from the target QoS. The transfer function of the system is represented as $P(z) = QoS_{max}/z$ which implies that, depending on the controller output, the QoS for the next control interval is some fraction of the maximum QoS. The global transfer function of the control system is $G(z) = \frac{C(z)P(z)}{1+C(z)P(z)}$ which, we enforce to be $\frac{1-p}{z-p}$. We substitute the controller function $C(z)$ as $\frac{z(1-p)}{Q_{max}(z-1)}$. Applying inverse z-transform on $C(z)$, we compute the discrete-time controller function, which quantifies the correspondence between the error term and the controller output (i.e., the next performance state). The result is the proportional integral (PI) controller representation shown in Equation 1. The pole (p) of global transfer function should be in range $[0, 1)$ to ensure stability and avoid oscillatory behaviour [14]. The value of p also allows to tradeoff robustness for responsiveness [14] and smaller values increase the controller's response to workload variations. We manually tune the value of p to be 0.4 on our system. The controller ensures convergence to target QoS as the steady state gain equals 1 ($G(z = 1) = 1$).

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

4. EXPERIMENTAL RESULTS

This section evaluates QScale for its effectiveness in maximizing durations of target QoS levels. In addition to the default *Interactive governor* [1] and *HMP scheduler* [4] pair, we also compare QScale to a *DVFS-only* policy. *DVFS-only* policy performs only closed-loop DVFS to deliver target QoS levels and uses the default HMP scheduling framework as opposed to the proposed thermally-efficient thread mapping.

Evaluation methodology: We evaluate policies under 3 target QoS levels for each application, corresponding to *high*, *medium* and *low* performance, determined based on how much QoS degradation is observed (Figure 1a). For instance, we omit the 70% target QoS case if throttling does not incur degradation below 70% of the maximum QoS when using the default management. We run each application up to 13 minutes. This duration provides long enough execution to cause thermal throttling in all QoS levels and allows us to determine the exact sustainable duration before the thermal headroom is fully exhausted. For QScale and *DVFS-only*

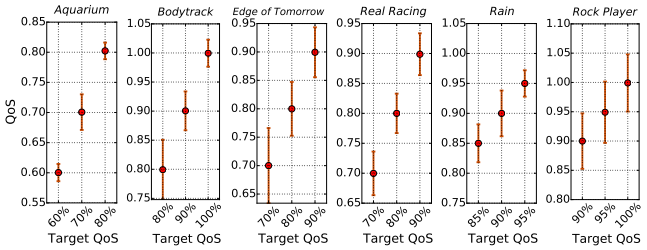


Figure 10: Average and standard deviation of QoS when using QScale under different target QoS levels.

policies, sustained QoS duration is the same as the overall duration before reaching thermal threshold where policies are able to maintain target QoS levels. For the default management, we report the time QoS was above each target level throughout the execution. We measured the maximum temperature as 59°C when idle. To achieve consistent temperature measurements, before each experiment, we cool the system below to 59°C using the fan and leave the platform idle for 15 minutes in order for temperature to stabilize.

Meeting QoS targets: Figure 10 demonstrates QScale's ability to meet the target QoS levels for each application. The figure plots average QoS and standard deviation for different QoS targets. While QScale meets the target QoS levels with only 3.8% deviation on average, we observe higher variation in specific applications. Gaming workloads (*Edge of Tomorrow, Real Racing*) and video player (*Rock Player*) incur higher deviation (6.6% in the worst case) as such applications have high dynamism due to scene changes and respective sudden variation in the processing requirements.

Extending sustainable QoS with QScale: Figure 12 demonstrates the sustained QoS durations. Sustained QoS durations increase as we lower the QoS requirements (left to right). This is intuitive as *DVFS-only* and QScale policies can lower the frequency and operate for longer durations without causing thermal throttling. The default policy also provides longer durations above the target QoS levels as it takes longer time for QoS to degrade to lower levels. QScale consistently provides the highest durations compared to both default management and *DVFS-only* policy.

For *aquarium*, we observe the shortest sustained durations. For instance, even using QScale at 60% target QoS, we are able to sustain this level only for 200 seconds. This application has the highest power consumption (1.18W CPU and 1.2W GPU average power at 60% QoS) and quickly exhausts the thermal headroom, leading to aggressive thermal throttling and QoS loss. *bodytrack* provides higher gains in sustained durations for the higher two QoS targets. For instance, while the default management and the *DVFS-only* policies cannot sustain 100% and 90% QoS levels for more than 15 and 130 seconds respectively, QScale delivers these QoS targets for 125 and 330 seconds (more than 8x and 2.5x longer). To illustrate the insight behind this result, in Figure 11, we show QoS and power at different big cluster DVFS levels when HMP scheduler or criticality-aware mapping is enabled. By moving the

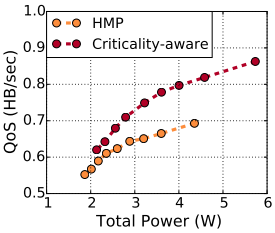


Figure 11: QoS/power scaling for *bodytrack*.

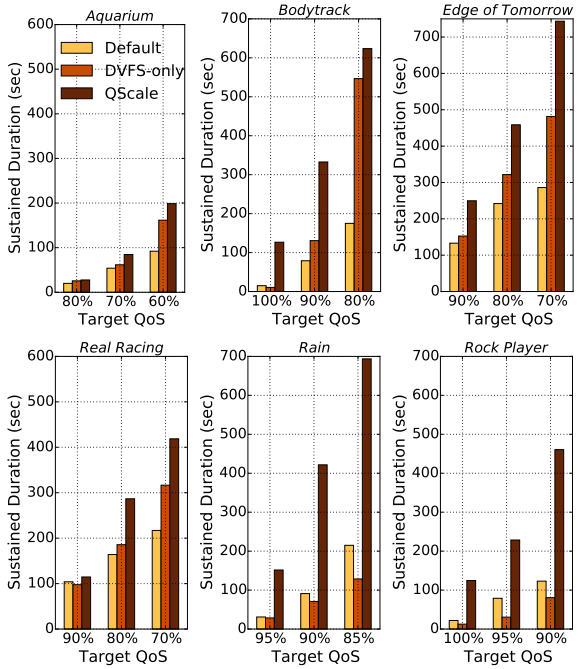


Figure 12: Sustained QoS durations with default management (*Interactive Governor + HMP scheduler*), *DVFS-only* and QScale policies under different QoS targets.

threads that bring the most QoS gains onto big cluster, QScale delivers the maximum QoS levels achieved using the HMP scheduler at much lower big core frequencies. For the two gaming applications, *Edge of Tomorrow* and *Real Racing*, QoS is dominated by a single thread with a relatively high CPU usage as shown in Figure 6 and Figure 7. QScale and HMP scheduler only schedules this thread to the big cluster, which is indicated by the similar usage of big cluster (22%-25% for both games). Therefore, the benefits of QScale are primarily due to thermally-efficient selection of core for execution. QScale achieves distinctively higher improvements across all QoS configurations for *Rain* and *Rock Player*. In these cases, QScale leverages the thread criticality information generated via offline thread characterization and reserves the big cores only for the QoS-critical threads, thus preventing the power-hungry big cores from quickly exhausting the thermal headroom.

Figure 13 illustrates QoS and maximum chip temperature traces for the *Rain* application when target QoS level is set to 90% of the maximum. While the default policy initially provides slightly (5%) higher QoS, it reaches the thermal limit immediately and leads to QoS degradation due to thermal throttling. For the *DVFS-only* policy, QoS starts to degrade after 95 seconds as the baseline scheduler greedily moves the non-critical application threads with high CPU usage onto big cluster. QScale schedules only the QoS-critical threads on the big cores in addition to controlling the DVFS, and extends the sustainability of the target 90% QoS level up to 440 seconds. In addition, QScale monitors the GPU power and assigns the threads onto the thermally-efficient big cores. In this example, Core-0 and Core-3 are prioritized for execution as seen by their higher utilization in the bottom left plot in Figure 13.

Figure 14 shows the QoS and temperature traces for *Rock Player* under a desired QoS level of 100%. While the *DVFS-only* policy starts throttling shortly after the 100% QoS re-

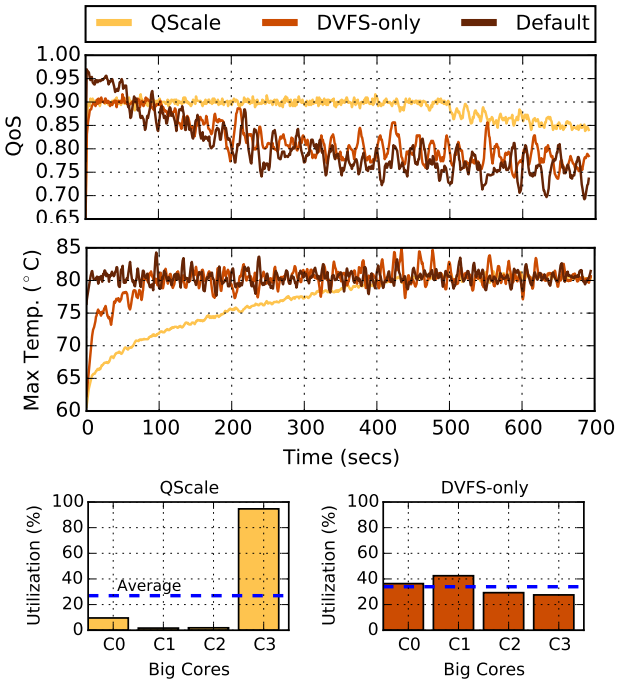


Figure 13: QoS and temperature traces of Rain application for the *default management*, *DVFS-only* and QScale policies for a target QoS level of 90%. Bottom figures show the average per-core utilization for QScale and *DVFS-only* policies.

quirement is met, QScale can deliver this maximum QoS target for two minutes by preventing non-critical application threads from greedily executing on the power-hungry cores and quickly elevating temperature. For this case, QScale achieves a 19% lower average big cluster utilization, as compared to the 47% utilization of the *DVFS-only* policy.

Adapting to dynamic QoS targets: QoS requirements may be altered during the application execution due to various reasons such as changes in user preferences, low battery, or low thermal headroom. We demonstrate QScale's ability to dynamically respond to changes in QoS requirements by modulating the target QoS level during the execution of the *Edge of Tomorrow* game. Figure 15 presents the QoS trace from this experiment and shows that QScale closely tracks dynamically altered target QoS levels.

5. RELATED WORK

Energy and Thermal Management: The idea of trading off accuracy or QoS with power appears in several prior energy management methods. PowerDial [16] elastically performs accuracy tradeoffs by dynamically tuning the application parameters under power caps. Zhu et al. [31] propose a runtime framework that trades off response time in latency-sensitive mobile web applications for energy savings.

Most prior work has focused on energy and reliability optimization on systems with *homogeneous CPU cores*. Kadjo et al. [19] propose a joint CPU-GPU control policy that minimizes DVFS levels during frequency insensitive phases of mobile games while delivering the maximum QoS. Das et al. [9] propose a reinforcement learning algorithm that controls CPU affinity and DVFS levels to simultaneously optimize peak, average temperature and thermal cycling. Sahin et al. [27] propose a closed-loop DVFS policy to minimize

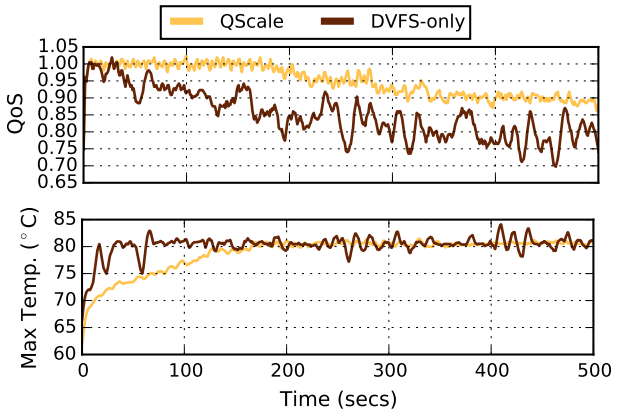


Figure 14: QoS and temperature traces of the Rock Player multimedia player application with *DVFS-only* and QScale policies for maximum target QoS level (100%).

temperature while meeting a given QoS constraint. To the best of our knowledge, there is no prior work that addresses the QoS-temperature tradeoffs achieved via scheduling and DVFS decisions on a real-life *heterogeneous* mobile CPU.

Some earlier work has conducted thread criticality analysis to guide DVFS on homogeneous multi-core systems (e.g., [7]). Our work, in contrast, determines critical threads in terms of their potential benefits when running on big cores.

Scheduling and Energy Management on Heterogeneous CPUs: Some of the prior scheduling techniques maximize overall throughput on heterogeneous multi-cores running multi-program workloads. Koufaty et al. [21] dynamically monitor several hardware events to guide load balancing decisions in the Linux scheduler. Other work [22] relies on application profiling on all core types to guide scheduling. On a big.LITTLE mobile platform, Hsiu et al. [17] achieve energy savings by providing more CPU resources to foreground applications while leveraging the little core cluster for background applications. Our work focuses on single foreground application scenario but identifies the heterogeneity within the application threads, which we use to perform thermally-efficient scheduling.

Pricopi et al. [24] propose a real-life power budgeting framework on a big.LITTLE system where target QoS of multiple single-threaded self-adaptive applications [15] are adjusted reactively. Various application-specific energy management policies have been proposed for heterogeneous mobile CPUs. Zhu et al. [32] analyze web-page features to determine DVFS settings while meeting latency constraints. Pathania et al. [25] derive offline performance estimation heuristics to guide big/LITTLE core allocation for multi-threaded mobile games and achieve energy savings without impacting the peak user experience.

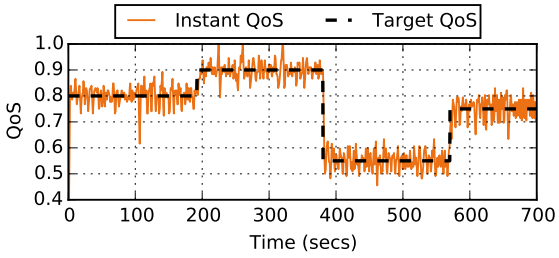


Figure 15: Adapting to dynamic QoS targets with QScale while running the *Edge of Tomorrow* gaming application.

Thermal Management on Heterogeneous CPUs: There has been relatively limited prior work that study the single-ISA heterogeneous CPUs from a temperature perspective. Sharifi et al. [29] propose a job allocation strategy for temperature balancing on a heterogeneous SoC to mitigate negative effects of thermal variations. Kim et al. [20] propose mDTM, which alternates between the peak performance and little core operation to allow for longer time spent at the highest performance state. Their technique shortens the overall execution time for CPU-bound applications with high frequency scalability. Singla et al. [30] provide a thermal modeling methodology using a real-life big.LITTLE platform and present a proactive DTM policy to prevent thermal violations. Our paper maximizes the duration before a thermal violation occurs by providing thermally-efficient QoS management. In a study which points to CPU-GPU thermal coupling effects in AMD's fused architectures, Paul et al. [26] limit the maximum CPU frequency to leave larger thermal headroom for boosting GPU power. In contrast, QScale recognizes such coupling effects to make thermally-efficient CPU core allocation decisions in low-TLP mobile applications.

This paper differentiates from the prior work as follows: (1) We show that greedily utilizing power-hungry cores significantly degrades user experience over extended durations due to thermal throttling; (2) we demonstrate the dependence of thermally-efficient core allocation decisions on dynamic CPU-GPU thermal couplings; (3) we show that QoS is dominated by a few QoS-critical threads and leverage this observation for thermally-efficient scheduling; (4) we propose a coordinated DVFS and thread allocation policy that simultaneously caters to both hardware and application heterogeneity.

6. CONCLUSION

This paper identifies the throttling-induced performance drawbacks of current DVFS and scheduling policies in mobile platforms with heterogeneous CPUs. We show that greedily maximizing performance under thermal restrictions using big cores can provide short bursts of high QoS, *but at the expense of significant QoS degradations over extended durations*. The proposed QScale policy provides *thermally-efficient QoS management* for mobile applications. By restricting the use of high-performance cores to a relatively low number of *QoS-critical threads* and monitoring the CPU-GPU thermal coupling for thermally-efficient core allocation, QScale slows down heating and provides users with up to 8x longer durations of *sustainable QoS*.

7. REFERENCES

[1] Android cpu governors. http://goo.gl/Mlr9U1.

[2] Aquarium. http://goo.gl/2RCoM4.

[3] big.LITTLE Technology. http://goo.gl/lMp9QV.

[4] Heterogeneous multi-processing. https://goo.gl/FaYRZG.

[5] Rain. https://goo.gl/5ReAJC.

[6] When benchmarks aren't enough: Cpu performance in the nexus 5. [online] http://goo.gl/ao4kAT.

[7] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[9] A. Das et al. Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems. In *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[10] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using latency to evaluate interactive system performance. *SIGOPS Oper. Syst. Rev.*, Oct. 1996.

[11] C. Gao, A. Gutierrez, M. Rajan, R. Dreslinski, T. Mudge, and C.-J. Wu. A study of mobile device utilization. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 225–234, March 2015.

[12] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *35th International Conference on Software Engineering (ICSE)*, pages 72–81, 2013.

[13] 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 *IEEE 22th International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[14] J. L. Hellerstein et al. *Feedback control of computing systems*. John Wiley & Sons, 2004.

[15] H. Hoffmann et al. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *International Conference on Autonomic Computing*, pages 79–88, 2010.

[16] H. Hoffmann et al. Dynamic knobs for responsive power-aware computing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–212, 2011.

[17] 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. Embed. Comput. Syst.*, 15(1):17:1–17:22, Jan. 2016.

[18] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 349–366. ACM, 2015.

[19] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz. A control-theoretic approach for energy efficient cpu-gpu subsystem in mobile platforms. In *DAC*, 2015.

[20] Y. G. Kim, M. Kim, J. M. Kim, and S. W. Chung. M-dtm: Migration-based dynamic thermal management for heterogeneous mobile multi-core processors. In *Design, Automation & Test in Europe (DATE)*, 2015.

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

[22] 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 *ISCA*, 2004.

[23] T. Lanier. ARM Cortex-A15 processor. http://goo.gl/0ARnm7.

[24] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin. Hierarchical power management for asymmetric multi-core in dark silicon era. In *DAC*, 2013.

[25] A. Pathania, S. Pagani, M. Shafique, and J. Henkel. Power management for mobile games on asymmetric multi-cores. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 243–248, July 2015.

[26] I. Paul, S. Manne, M. Arora, W. L. Bircher, and S. Yalamanchili. Cooperative boosting: Needy versus greedy power management. In *ISCA*, 2013.

[27] O. Sahin, P. Varghese, and A. Coskun. Just enough is more: Achieving sustainable performance in mobile devices under thermal limitations. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. ACM, 2015.

[28] 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 *IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–11, Oct 2015.

[29] S. Sharifi, A. Coskun, and T. Rosing. Hybrid dynamic energy and thermal management in heterogeneous embedded multiprocessor socs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 873–878, 2010.

[30] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras. Predictive dynamic thermal and power management for heterogeneous mobile platforms. In *DATE*, pages 960–965, 2015.

[31] Y. Zhu, M. Halpern, and V. J. Reddi. Event-based scheduling for energy-efficient QoS (eQoS) in mobile web applications. In *HPCA*, pages 137–149, 2015.

[32] Y. Zhu and V. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA*, 2013.