BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

EFFICIENT RUNTIME MANAGEMENT FOR ENABLING SUSTAINABLE PERFORMANCE IN REAL-WORLD MOBILE APPLICATIONS

by

ONUR SAHIN

B.S., Istanbul Technical University, 2013

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2019

Approved by

First Reader

Ayse K. Coskun, Ph.D.
Associate Professor of Electrical and Computer Engineering

Second Reader

Manuel Egele, Ph.D.
Assistant Professor of Electrical and Computer Engineering

Third Reader

Michel Kinsy, Ph.D.
Assistant Professor of Electrical and Computer Engineering

Fourth Reader

Ajay Joshi, Ph.D.
Associate Professor of Electrical and Computer Engineering

The important thing is not to stop questioning. Curiosity has its own reason for existence. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvellous structure of reality. It is enough if one tries merely to comprehend a little of this mystery each day. Never lose a holy curiosity.

Albert Einstein

Acknowledgments

First, I thank Prof. Ayse Coskun for all her help and support throughout my PhD. Her encouragement allowed me to turn much of this work into papers and present at multiple conferences. I also owe a great deal of debt to Prof. Manuel Egele. His emphasis on clarity and attention to detail have had a profound impact on the way I think, approach and speak about technical problems. I also thank my committee members, Profs. Michel Kinsy and Ajay Joshi, for their time and feedback.

I would like to thank my parents and brother who gave me their unwavering love and cared for my well-being before anything else. While I am deeply grateful to many of my relatives who helped in many ways, I would specifically like to acknowledge several. I thank my uncle, Gursel, and his lovely family who have supported and believed in me. I also thank my cousin, Ekrem, for encouraging me into engineering (and still teasing me for not studying CS :)).

I would also like to thank my friends for their unquestionable role in maintaining my sanity, and brining fun and joy into my graduate life: Ozan Tuncer, Tolga Bolukbasi, Onur Zungur, Tiansheng Zhang, Can Hankendi, Ozan Tezcan, Golsana Ghaemi, Asselya Aliyeva, Furkan Eris, Cantay Caliskan, Celalettin Yurdakul, Ata Turk, Fulya Kaplan, Prachi Shukla, Zihao Yuan, Aditya Narayan, Emre Ates, Mert Toslali, Burak Aksar and Yijia Zhang. My appreciation also goes to the members of ICSG, CAAD and SeclaBU labs, with whom we shared many moments as well.

I thank Darrin Johnson for the internship opportunity at Oracle and giving me the freedom to work on projects I personally found interesting. I also thank Sridhar Sundaram and Meeta Srivastav for their invaluable mentorship during my internship at Samsung and providing me with an enjoyable and productive working environment.

EFFICIENT RUNTIME MANAGEMENT FOR ENABLING SUSTAINABLE PERFORMANCE IN REAL-WORLD MOBILE APPLICATIONS

ONUR SAHIN

Boston University, College of Engineering, 2019

Major Professors: Ayse K. Coskun, Ph.D.
Associate Professor of Electrical and Computer Engineering

Manuel Egele, Ph.D.
Assistant Professor of Electrical and Computer Engineering

ABSTRACT

Mobile devices have become integral parts of our society. They handle our diverse computing needs from simple daily tasks (i.e., text messaging, e-mail) to complex graphics and media processing under a limited battery budget. Mobile system-on-chip (SoC) designs have become increasingly sophisticated to handle performance needs of diverse workloads and to improve user experience. Unfortunately, power and thermal constraints have also emerged as major concerns. Increased power densities and temperatures substantially impair user experience due to frequent throttling as well as diminishing device reliability and battery life. Addressing these concerns becomes increasingly challenging due to increased complexities at both hardware (e.g., heterogeneous CPUs, accelerators) and software (e.g., vast number of applications, multi-threading). Enabling *sustained user experience* in face of these challenges requires (1) practical runtime management solutions that can reason about the performance needs of users and applications while optimizing power and temperature; (2) tools for analyzing real-world mobile application behavior and performance.

This thesis aims at improving sustained user experience under thermal limitations by incorporating insights from real-world mobile applications into runtime management. This thesis first proposes thermally-efficient and Quality-of-Service (QoS) aware runtime management techniques to enable sustained performance. Our work leverages inherent QoS tolerance of users in real-world applications and introduces QoS-temperature tradeoff as a viable control knob to improve user experience under thermal constraints. We present a runtime control framework, QScale, which manages CPU power and scheduling decisions to optimize temperature while strictly adhering to given QoS targets. We also design a framework, Maestro, which provides autonomous and application-aware management of QoS-temperature tradeoffs. Maestro uses our thermally-efficient QoS control framework, QScale, as its foundation.

This thesis also presents tools to facilitate studies of real-world mobile applications. We design a practical record and replay system, RandR, to generate repeatable executions of mobile applications. RandR provides this capability by automatically reproducing non-deterministic input sources in mobile applications such as user inputs and network events. Finally, we focus on the non-deterministic executions in Android malware which seek to evade analysis environments. We propose the Proteus system to identify the instruction-level inputs that reveal analysis environments.

Contents

List of Tables

List of Figures

List of Abbreviations

AF	......	Android Framework
API	......	Application Programming Interface
App	......	Application
ART	......	Android Runtime
CMOS	......	Complementary metal oxide semiconductor
CPU	......	Central Processing Unit
DPM	......	Dynamic Power Management
DSP	......	Digital Signal Processor
DTM	......	Dynamic Thermal Management
DVFS	......	Dynamic Voltage and Frequency Scaling
FFT	......	Fast Fourier Transform
GHz	......	Gigahertz
GPU	......	Graphics Processing Unit
I/O	......	Input and Output
IP	......	Intellectual Property
ISA	......	Instruction Set Architecture
OS	......	Operating System
QoS	......	Quality of Service
SoC	......	System-on-a-Chip
SSL	......	Secure Sockets Layer
TLS	......	Transport Layer Security
UI	......	User Interface
VM	......	Virtual Machine

Chapter 1

Introduction

Mobile devices provide us with the essential computing power to run applications that improve our quality of life. These applications serve our daily necessities (e.g., e-mail, cellular communication) while also providing entertainment (e.g., gaming, social networking, etc.). A major goal over generations of mobile system design has been to improve user experience. Traditionally, this objective has been fulfilled by allowing computationally complex applications to run with higher performance under limited energy budgets (Halpern et al., 2016). Such mainstream design goals, along with the competitive mobile market, has lead to aggressive mobile system-on-a-chip (SoC) designs. State-of-the-art mobile SoCs comprise high-performance multicore CPUs and various integrated accelerators (e.g., GPU, DSP etc.) to provide the best performance for users (Lanier, 2017).

Power dissipation of such high performance SoCs, however, can significantly elevate under increased computational demand. Thus, limited battery life and thermal constraints became two major roadblocks in further extending the user experience [1] (Halpern et al., 2016). Since the elevated temperatures trigger performance throttling mechanisms to prevent thermal violations, *sustaining performance at user-acceptable levels* becomes increasingly challenging. Such thermally-induced performance losses have already become major sources of performance loss for mobile users (Ho and Frumusanu, 2014). Therefore, it is essential to have techniques that can adaptively tune

[1] In this thesis, we use the term user experience in the context of performance perceived by the end user.

the power consumption of major power-hungry components (e.g., CPU) at runtime to minimize energy and temperature without violating user performance requirements.

Current OS-level runtime management policies greedily increase CPU resources to improve performance under increased computational demand. These policies often rely on coarse-grained usage metrics to determine the computational demand. While such policies are widely deployed across mobile, desktop or server systems, cooling and battery restrictions deem them largely inefficient for mobile devices. Greedily improving performance does not always lead to improved mobile user experience as the performance may exceed the capabilities of human perception (Zhu et al., 2015a). In fact, user experience may deteriorate due to thermal throttling caused by increased power consumption and excessive heat generation (Ho and Frumusanu, 2014), which cannot be easily dissipated due to inherent cooling limitations of mobile platforms. Mobile applications also widely differ from traditional CPU benchmarks or server-class workloads in terms of their user interface (UI) -driven asynchronous execution model and workload behavior (e.g., thread-level parallelism (Gao et al., 2014), phase behavior (Zhu et al., 2015a)). Current policies in system software, however, are traditionally optimized using CPU/GPU benchmarks or workloads that do not completely represent the characteristics of real-world mobile applications (Huang et al., 2014; Pandiyan et al., 2013). As a result, such policies often perform inefficiently on typical mobile applications in the wild. Our key observation behind these drawbacks, which motivates this thesis, is that *the current runtime management policies are designed with limited insight from the actual real-world mobile applications and work with limited feedback from users, application, or the underlying platform.*

This thesis claims that the current runtime policies that greedily exhaust thermal headroom for performance cause significant QoS loss over extended durations and sustained QoS can be substantially improved with a runtime framework which

1) enforces QoS tradeoffs proactively in an application-specific and thermally-aware manner; 2) systematically incorporates insights from real-world mobile applications. To this end, this thesis proposes runtime management policies that leverage insights from the behavior of real-world mobile applications. We also provide software frameworks to facilitate studying the characteristics of mobile applications for improving mobile system efficiency and security.

1.1 Runtime Management for Sustainable Performance

The challenge in providing energy- and thermally-efficient operation in mobile devices rises due to various factors. First, due to indisputable significance of performance on user experience, any decision by runtime power/thermal management policies should take into account the strict quality-of-service (QoS)[2] requirements needed by the user. As the diversity in computation requirements of mobile applications grows, single-ISA heterogeneous multi-core architectures also gained popularity. Heterogeneous CPUs provide more energy-efficient operation (Kumar et al., 2004; Kumar et al., 2003) due to their wide dynamic power and performance ranges. Due to such asymmetries in hardware and respective changes in power/performance/temperature tradeoffs, the complexity of runtime management decisions increases substantially. In addition, tight integration of high power accelerators such as GPUs along with CPUs can widely alter chip temperatures at runtime. Hence, analyzing and optimizing the temperature of individual processors (i.e., CPU or GPU) in isolation becomes both challenging and inefficient (Prakash et al., 2016; Sahin and Coskun, 2016b).

In this thesis, we aim at providing the users with *sustainable performance levels over extended durations* under thermal constraints (i.e., in contrast to always maximizing performance). We argue that runtime power/thermal management strategies

[2]In this thesis, QoS refers to a metric used to quantify the performance experienced by the user (e.g., frames-per-second (FPS) or response latency).

should take into account the QoS requirements of users and make conservative use of thermal headroom. Such a strategy is necessary to provide users with consistent performance when they interact with their phones over extended durations (as in gaming and streaming). This is in contrast to the existing runtime strategies which greedily exhaust the available thermal headroom to boost QoS when computational demand increases. This work demonstrates, for the first time, that such an approach can significantly hurt QoS over extended durations due to more aggressive throttling that needs to be applied to control maximum temperatures (Chapter 2). We propose to achieve sustained performance via 1) novel runtime management techniques that can optimize power and temperatures under strict QoS requirements; 2) application-aware QoS management techniques that can autonomously and proactively manage QoS. The specific contributions are as follows:

- We demonstrate up to 50% QoS loss with existing runtime policies due to always maximizing short-term performance (Sahin and Coskun, 2015). Thus, we propose a runtime dynamic voltage and frequency scaling (DVFS) scheme for thermally-efficient QoS tradeoffs (Sahin et al., 2015) (Section 4.1). Our technique dynamically monitors application-specific QoS to adjust DVFS levels using a closed-loop control system. To improve the thermal efficiency of our closed-loop QoS control, we also propose a DVFS state scheduling scheme. Our DVFS state scheduler temporally distributes the high power discrete DVFS states to minimize the peak temperature without violating performance targets.

- We propose a runtime framework, QScale (Sahin and Coskun, 2016b), that lowers temperatures under performance guarantees on state-of-the-art heterogeneous multicore mobile CPUs (Section 4.2). In addition to closed-loop DVFS control, QScale efficiently guides mapping of application threads onto “big” and “little” cores. During mapping, QScale considers the *on-chip CPU-GPU* *thermal couplings* as well as taking into account the *QoS-criticality* of individual application threads. QScale improves sustained QoS durations by up to 8x as compared to existing DVFS and scheduling approaches that rely on coarse-grained utilization metrics to guide power management decisions and disregard CPU-GPU thermal interactions.

- We design a framework, Maestro (Sahin et al., 2018), to autonomously guide QoS-temperature tradeoffs (Section 4.3). Maestro considers the bursty and throughput-oriented computation characteristics of mobile applications to guide QoS-temperature tradeoffs. Upon projecting throttling-induced QoS degradations, Maestro proactively lowers QoS to improve sustained performance.

1.2 Software Frameworks for Studying Real-World Applications

Mobile applications widely differ from traditional CPU benchmarks (e.g., SPEC CPU (Henning, 2006), PARSEC (Bienia et al., 2008)) in terms of their workload characteristics as well as their interactive nature (Gao et al., 2015; Gutierrez et al., 2011). Studying and understanding the behavior of real-world mobile applications are crucial to optimize current system software and improve the efficiency of future mobile devices. A fundamental requirement while studying the characteristics of real-world mobile applications is the ability to reproduce their executions. For instance, from the perspective of OS-level policy optimization, the same execution of an application is often repeated under different scheduling or power management policies to explore energy and performance tradeoffs. Reproducing the executions of real-world mobile applications, however, has proven difficult under real-life scenarios due to various non-deterministic factors such as user inputs, network or sensory input.

Due to the inability to easily reproduce real-life behavior of mobile applications, many characterization studies analyze mobile workloads under limited usage scenarios (e.g., only application launch) (Huang et al., 2014). Such studies provide only limited insight into mobile workload characteristics while designing runtime policies. Other studies rely on hand-crafted test scripts for specific devices and applications (Li et al., 2017). However, such application- or device-specific test cases cannot be easily reproduced across different systems, diminishing the reproducibility of scientific outcomes. We argue that systematic understanding of mobile workloads and comparison of experimental observations require the availability of software frameworks to reproduce realistic executions in a cross-platform manner.

Benign input-dependent (e.g., UI, network) execution variations are not the only source of non-deterministic behavior in mobile applications. Unfortunately, recently surging *evasive Android malware* behave non-deterministically (i.e., alter their behavior) based on the environment they are executed in. By ceasing malicious activities in testing environments, such malware seeks to evade detection by malware analyzers (Xu, 2017; Davis, 2017; Branco et al., 2012). A crucial step for defending against such malware is to systematically extract the discrepancies between the testing environments (e.g., an emulator) and real systems. Once discovered, such discrepancies can be proactively eliminated (Liu et al., 2017) or can be used to inspect applications for presence of evasion tactics (Branco et al., 2012).

This thesis presents software frameworks to facilitate the studies involving real-world mobile applications. Our specific contributions are as follows.

- We design a record and replay framework for Android, RandR (Sahin et al., 2019). RandR enables repeatable executions of mobile applications by capturing and replaying UI and network inputs in a practical and cross-platform manner (Section 5.1). RandR can be deployed on commodity mobile platforms. RandR achieves such practical merits through a novel dynamic instrumentation based approach which eliminates the need for any static OS/application instrumentation, privileged mode or any specialized hardware support.

- We propose the Proteus system (Sahin et al., 2018) to automatically extract instruction-level discrepancies of emulated analysis environments. Such discrepancies can be leveraged by malware to distinguish the underlying environment and evade analysis by ceasing malicious behavior (Section 5.2). Proteus collects and automatically analyzes a large number of instruction-level traces from real-life ARM CPUs as well as from an instrumented emulator to pin-point instruction-level discrepancies.

Chapter 2

Problem Statement and Motivation

Real-world interactive mobile applications widely differ from traditional CPU benchmarks in terms of their workload behavior and performance requirements. Current mobile systems that execute our daily real-world applications are also heavily constrained in terms of power and cooling. To bridge the gap between real-world applications and runtime management, this thesis makes the case for leveraging insights from real-world application behavior to improve mobile system efficiency and security. We first demonstrate the drawbacks of existing greedy thermal management policies in mobile systems to motivate our approach of incorporating QoS requirements into runtime management (Section 2.1). Next, we discuss the challenges involved in characterization of real-world applications (Section 2.2).

2.1 (Un)sustainable Performance in Mobile Platforms

Traditionally, thermal management research has focused on maximizing performance under thermal restrictions (Bartolini et al., 2011; Chantem et al., 2009; Singla et al., 2015), assuming that such an approach will always maximize the user experience. The unique observation that motivates our work in this thesis contradicts this assumption: *greedily exhausting the thermal headroom improves short-term user-experience, but at the expense of significantly less QoS over extended durations.* Section 2.1.1 illustrates this observation through experiments with various mobile applications running on real-life mobile platforms. We demonstrate a case of *QoS-temperature tradeoff* where user experience can be improved by trading off short-term QoS to gain extra thermal headroom that could be used to extend the durations of *sustained performance*. Section 2.1.2 illustrates this intuition using real-life examples. Finally, in order to guide such QoS-temperature tradeoffs autonomously, we demonstrate the need for considering the unique characteristics of real-world mobile applications (Section 2.1.3).

2.1.1 Impact of Thermal Constraints on Performance

Modern mobile devices incorporate thermal throttling strategies that react to elevated chip and skin level temperatures by slowing down the CPU (e.g., via DVFS or power gating of individual cores). These policies reduce the power dissipation and keep temperatures below safe thresholds. Greedily exhausting thermal headroom to maximize QoS can improve user experience if throttling does not happen frequently. However, our research has revealed that rapidly elevated on-chip and external component (i.e., battery, display etc.) temperatures can lead to increasingly aggressive thermal throttling over time and cause as much as 50% QoS loss (Sahin and Coskun, 2015; Sahin et al., 2015). Such a large throttling-induced QoS loss over long durations of use (as in gaming and streaming) becomes a crucial problem in mobile devices, which are inherently limited by cooling capabilities. Mobile users expect consistent and acceptable QoS while running applications for minutes or longer. Users have, in fact, already reported significant performance loss and dissatisfaction during extended durations of device use (Cunningham, 2013; Ho and Frumusanu, 2014).

We illustrate this problem of unsustainable performance through experiments on two state-of-the-art mobile platforms. Consider the case in Figure 2·1 which shows the percentage of time spent in different frequency levels over time on a Snapdragon MDP8974 smartphone platform (Snapdragon MSM8974 MDP., 2014) while running a common FFT kernel (Pozo and Miller, 2000) repeatedly. The default Linux CPU

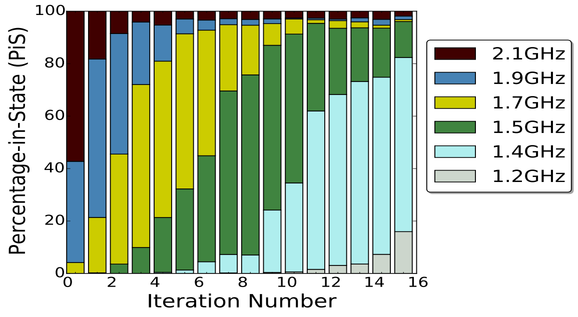


Figure 2·1: Frequency residencies over time on a MDP8974 smartphone during continuous use. Performance impact of throttling increases over time as the CPU is forced to use lower frequencies to meet the thermal constraints.

power manager [3] scales the frequency to the highest level to boost performance and, initially, the application is able to operate below the thermal limit by throttling down to lower two frequencies only (1.9-1.7GHz). It should be noted that, in this example, frequencies lower than 2.1GHz level are enforced due to thermal throttling rather than the power management scheme. In the later iterations, there is a clear shift towards utilizing lower frequencies, which significantly reduces performance over time. This is due to more aggressive throttling applied by the baseline thermal management policy [3]. For instance, in the last iteration #16, more than 80% of the running time is spent at 1.4GHz and 1.2GHz, while the application was well able to run without scaling down to those two frequencies initially.

Figure 2·2 shows the QoS degradation over time for various real-life applications (involving gaming, media streaming) running on a Odroid-XU3 (Hardkernel, 2017) mobile platform. Odroid-XU3 platform integrates an emerging heterogeneous multi-core CPU architecture (i.e., big.LITTLE (ARM, 2013)). Over an 8-minute continuous use, throttling incurs significant QoS loss over time for all applications, reaching up to 50% degradations for the Aquarium application. While heterogeneous

[3]In the given experiment, the default ondemand governor in Linux is used. The baseline thermal throttling policy is a PID controller with a 80 °C thermal set-point that operates hierarchically with the ondemand governor.

CPU designs can significantly improve energy-efficiency (Pathania et al., 2015; Seo et al., 2015; Zhu and Reddi, 2013), clearly, thermal challenges continue to limit user-experience over long durations of device use. Overall, these examples illustrate that *greedily utilizing the thermal headroom to boost short term performance can lead to significant performance loss over extended durations.*

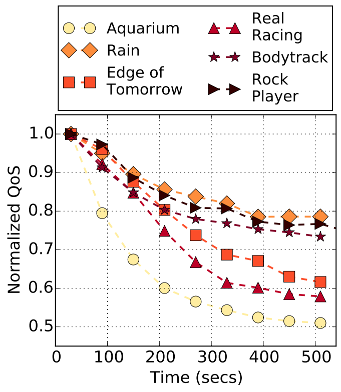
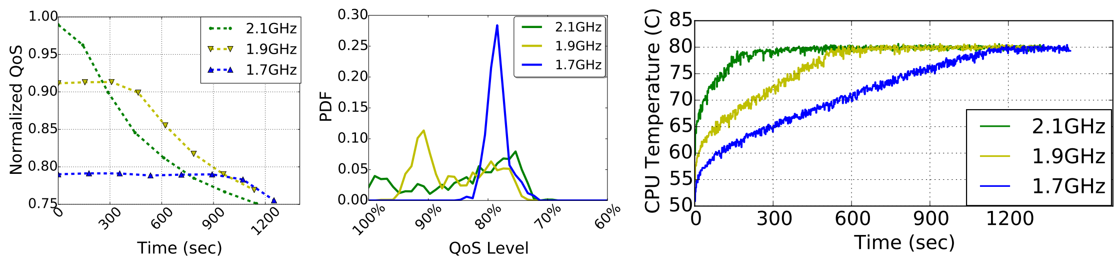


Figure 2·2: QoS degradation over time on Odroid-XU3 platform (Hardkernel, 2017).

2.1.2 A Case for Sustained QoS

Providing users with *longer durations of sustainable QoS* requires thermal management policies to deviate from existing greedy approach and adopt thermally-efficient strategies that can make more conservative usage of thermal headroom. In this section, we provide an experimental scenario to point to the potential trade-off between the instant short-term performance and sustainable performance. Our goal is to motivate thermally-efficient runtime management as well as the viability of QoS-temperature tradeoff to improve sustained performance.

Figure 2·3 presents an experiment that corresponds to a repetitive run of the Bodytrack application (Bienia et al., 2008) at three different static frequency levels. Note that the system can still throttle the frequency below the assigned static level to avoid thermal violation. Figure 2·3a shows the average QoS for each iteration of the application over time where the QoS is measured as heartbeats-per-second (Hoffmann et al., 2010). The maximum static frequency setting, 2.1GHz, gives the highest QoS initially. The QoS level, however, sharply reduces after the CPU reaches its thermal limits at around 220 seconds, as shown in Figure 2·3c. The QoS level continues to downgrade as a result of more aggressive thermal throttling and, at the end of the execution, QoS degrades to 25% of the initial maximum. The QoS drops



(a) QoS over Time **(b)** QoS Distribution **(c)** CPU Temperature Trace

Figure 2·3: An illustration of the effect of trading off the short term performance on performance sustainability. The experiment corresponds to a repetitive run of Bodytrack application (Bienia et al., 2008) at 3 static frequency settings and QoS values are normalized to maximum QoS.

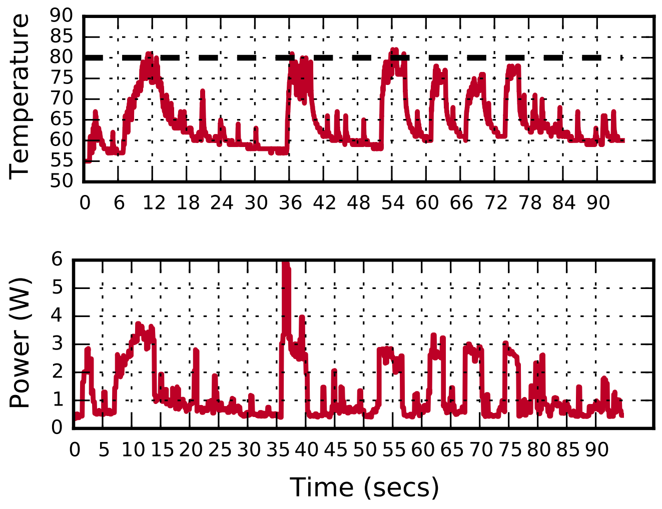
below 90%[4] at around 300 seconds when using the aggressive 2.1GHz setting. Setting the frequency at 1.9GHz frequency, however, allows to sustain the QoS level above 90% for 450 seconds. Figure 2·3b shows the QoS distribution for this experiment. The highest power setting results in wider distribution of the QoS while the 1.9GHz setting is able to rein the QoS distribution towards the 90% range (indicating longer duration of execution time spent around the 90% QoS). These results indicate that *lowering the short term performance requirements to "barely" meet the target QoS level, can enable longer sustainability of desired QoS goals.* This motivation forms the basis of our novel *QoS-centric thermal management* approach that we present in this thesis. The main objective of our techniques is to mitigate throttling-induced QoS degradations by slowing down the heating while delivering 'just enough' (as opposed to the best) QoS for meeting given user performance requirements (Sahin et al., 2015).

[4]We choose 90% as an example acceptable QoS level to explain our motivation.

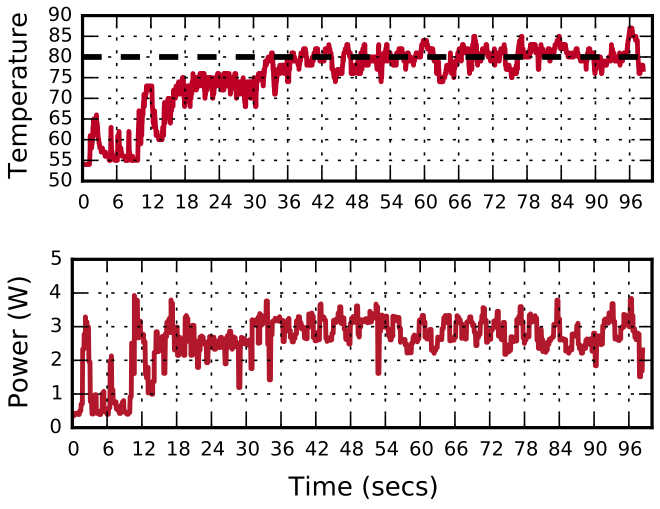
2.1.3 Need for Application Awareness in QoS Management

While QoS-temperature tradeoff can be a viable option for extending the sustained QoS durations, it is non-trivial 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 (Zhu et al., 2015a). 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., several minutes or larger), 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. Figure 2·4 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 (Figure 2·4a) 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, one can tolerate exhausting the 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 $t = 15s$. This continuous load causes a relatively stable power consumption ($\sim 3W$) and con-



(a)



(b)

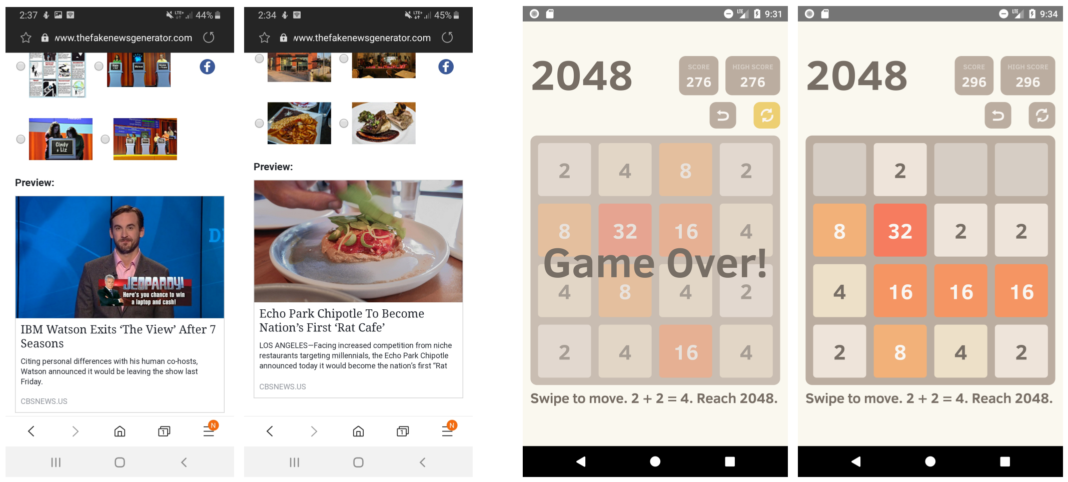
Figure 2·4: Distinct power and thermal profiles of a bursty application and a throughput-oriented graphics application. (a) *Adobe PDF Reader* application (b) *Rain* WebGL animation running in Chrome web browser. *Adobe PDF* application consists of bursts of computations such as zoom-in/out or text search 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 ($t = 18s$) in *Rain* application causes a relatively more steady power profile.

sistent increase in temperature. As shown in Figure 2·2, throttling incurs significant QoS loss due to continuous thermal violations and more aggressive 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.

2.2 Challenges of Characterizing Real-World applications

Current mobile systems research still primarily relies on workloads (e.g., benchmarks) that do not capture the real-world characteristics of mobile applications. Existing mobile benchmark suites (Gutierrez et al., 2011; Huang et al., 2014) provide only simplified use cases that do not fully represent real-world user behavior or capture only a few select domain of applications. Studying real-world behavior of mobile applications, however, can offer unique insights to optimize runtime management towards providing more sustainable performance under power and thermal limitations of contemporary mobile platforms (Park et al., 2016; Pandiyan et al., 2013). Unfortunately, real-world mobile applications present major challenges for real-life studies due to their non-deterministic nature. Unlike traditional benchmarks used for computer system evaluation and optimization (e.g., SPEC CPU2006 (Henning, 2006), PARSEC (Bienia et al., 2008)) that run to completion with minimal I/O, mobile applications are heavily input driven.

A common source of input that determine the application behavior is the UI inputs from the users in the form of touchscreen events. If not reproduced accurately across different runs of the same application, UI inputs can drastically alter the execution of an application. Such differences in executions would deem comparisons and observations across different executions inaccurate or not meaningful. In addition to UI inputs, other non-deterministic inputs such as network events or random numbers


www.thefakenewsgenerator.com
Preview:
2048
SCORE
HIGH SCORE
Game Over!
Swipe to move. 2 + 2 = 4. Reach 2048.

IBM Watson Exits 'The View' After 7 Seasons

Citing personal differences with his human co-hosts, Watson announced it would be leaving the show last Friday.

CBSNEWS.US

Echo Park Chipotle To Become Nation's First 'Rat Cafe'

LOS ANGELES—Facing increased competition from niche restaurants targeting millennials, the Echo Park Chipotle announced today it would become the nation's first "Rat

CBSNEWS.US

(a) (b)

Figure 2·5: Execution variations in various mobile applications due to various non-deterministic input sources. (a) The effect of network content changes. (b) Replay failure with Reran (Gomez et al., 2013) in 2048 application due to random numbers.

can also lead to different behavior in a given mobile application. For instance, Figure 2·5a shows the result of browsing the `fakenewsgenerator.com` website twice. The figure indicates different results in the visible state of the application due to changes in the network data. Prior studies have shown that different network content can widely alter power and performance tradeoffs due to variations in the compute requirements (Zhu and Reddi, 2013). Random numbers can also cause variant application behavior due to their inherent non-deterministic nature. As a result, existing tools that focus purely on reproducing UI inputs (Gomez et al., 2013; Halpern et al., 2015; Qin et al., 2016) may not be sufficient to enable repeatable executions. Figure 2·5b illustrates the experiment where we have recorded (left) and replayed (right) a set of UI interactions with an existing record and replay framework (i.e., Reran (Gomez et al., 2013)) for the popular 2048 gaming application. Due to random location of number that appear on the screen, Reran cannot reproduce the execution that have resulted in the "Game Over" text being rendered on the screen during record.

This thesis argues the need for a system with the following desirable characteristics to allow for reproducible studies with mobile applications. First, such a system should be should be easily extensible to handle multiple sources of inputs (e.g., UI, network) in contemporary mobile applications . Second, the executions should be easily reproduced across different platforms (i.e., cross-platform replayability). Finally, from a practical standpoint and ease of use, such a system should be compatible with existing hardware and software stack without requiring instrusive modifications.

Evasive Malware: The input sources such as UI or random numbers can intuitively result in different application behavior. However, an emerging class of evasive Android malware can present different behavior depending on a different information input: *whether the application is executed on an emulator or real system.* Use of emulators is currently the most dominant approach adopted by malware analysis tools in both academia (Yan and Yin, 2012; Tam et al., 2015) and industry (Oberheide and Miller, 2012). By detecting whether the application is executed in an emulated analysis environment or a real device, a malware can cease any malicious behavior under analysis and evade detection. Unfortunately, for Android, several recent families of evasive malware (e.g., *Xavier* (Xu, 2017), *Grabos* (Davis, 2017)) have already been identified in the Play Store. The focus of this thesis in this aspect is to systematically identify the information inputs that a malware can use to detect emulation. Such systematic identification is a crucial step towards understanding the behavior of mobile applications under real-world inputs and restoring the effectiveness of dynamic malware analysis systems.

Chapter 3

Background and Contributions

This thesis proposes runtime management techniques that leverage insights from the behavior of real-world mobile applications as well as providing frameworks to facilitate studying mobile applications. In the first section, we present a detailed overview of the state-of-the-art in runtime power and thermal management techniques. In Sections 3.2, we review prior techniques for enabling repeatable executions of real-world mobile applications. Section 3.3 provides an overview of the existing approaches for characterizing and preventing execution variations of evasive Android malware. Section 3.4 highlights the novel aspects of this thesis.

3.1 Runtime Management for System Efficiency

Reducing power and temperature has been long-standing objectives in design and management of computer systems. Power is the critical factor in determining the battery life and user experience for mobile devices in particular (Lee, 2014; Vince, 2014). Elevated temperatures degrade device reliability (EIA/JEDEC, 2016) and lead to performance loss due to throttling in mobile systems where active cooling is not viable (Singla et al., 2015). Thus, there has been a considerable amount of prior work in power and thermal management. This section provides an overview of the state-of-the-art in dynamic power and thermal management approaches proposed for traditional homogeneous CPUs (Section 3.1.1) as well as the techniques particularly focusing on heterogeneous multicore CPU designs (Section 3.1.2). Section 3.1.3 reviews the runtime policies that consider application-level feedback. We describe the techniques specifically targeting mobile platforms and applications as well as those that are broadly applicable to other computer systems (e.g., server and desktop PC).

3.1.1 Dynamic Power and Thermal Management of Homogeneous CPUs

Power Management with DVFS: Modern CPUs support multiple voltage/frequency states. Since the power consumption of CPUs with CMOS logic varies quadratically with voltage and linearly with frequency (Mudge, 2001), dynamic voltage and frequency scaling (DVFS) techniques have been proposed for power reduction. Such approaches reduce power by lowering voltage/frequency levels during frequency insensitive (e.g., memory or IO bound) phases of an application (Isci et al., 2006).

Since many user-centric applications require QoS guarantees (e.g., FPS in mobile games (Pathania et al., 2014; Kadjo et al., 2015), web page loading time in browsing (Zhu and Reddi, 2013)), various approaches consider power management under performance guarantees. Ayoub et al. (Ayoub et al., 2011) propose a closed-loop DVFS controller for meeting throughput requirements in a server system. Lo et al. (Lo et al., 2014) present PEGASUS, which utilizes the Intel's *Running Average Power Limiter* for enabling fine-grained CPU power tuning in web servers to match query latency requirements. Kadjo et al. (Kadjo et al., 2014) reduce the QoS requirements in memory bound applications and achieve platform level power savings in a mobile system. Pathania et al. (Pathania et al., 2014) propose a CPU-GPU power budgeting algorithm to meet a frames-per-second constraint in mobile games. While the above techniques do not consider the sustainability of performance targets and thermal impacts, a recent study points to a similar observation to ours (Section 2.1.1), emphasizing the impact of duration on measured performance on thermally-constrained systems (Emurian et al., 2014). Their work, however, focuses only on the performance measurement flaws that occur due to power level differences between only the boosting mode and the throttling mode in Intel's TurboBoost enabled processors.

Runtime QoS Tradeoffs: The idea of trading off accuracy or QoS with power appears in several prior energy management methods (Hoffmann, 2015). PowerDial (Hoffmann et al., 2011) elastically performs accuracy tradeoffs by dynamically tuning the application parameters under power caps to meet the performance goals. Zhu et al. (Zhu et al., 2015b) propose a runtime framework that trades off response time within a user tolerable range in latency-sensitive mobile web applications for energy savings. Trading off the QoS to proctively reduce temperature, on the other hand, is a novel insight brought by our work to address the QoS unsustainability problem in mobile devices over extended durations of use.

Thermal Management of Multi-core CPUs and Smartphones: Dynamic thermal management techniques are widely deployed and studied to improve energy efficiency and performance while ensuring safe chip (Skadron et al., 2003) and skin level temperatures (Sahin and Coskun, 2016a; Egilmez et al., 2015). Architecture-level approaches such as instruction fetch toggling (Skadron et al., 2003) or low-power pipelines (Lim et al., 2002) can provide fine-grained optimization of thermal hot spots while OS-level runtime management techniques can allow for design of more sophisticated control algorithms. Control-theoretic DVFS techniques provide effective temperature control while maximizing performance (Bartolini et al., 2011; Skadron et al., 2002). Predictive techniques have been applied to project thermal emergencies for minimizing temperature violations (Yeo et al., 2008). Such dynamic thermal management approaches are complementary to our techniques as they can be employed to minimize the QoS loss once the thermal headroom is fully exhausted.

There exists a body of work towards thermal modeling and management of mobile devices in particular. Xie et al. propose a resistance network based thermal simulation framework for obtaining component level steady-state temperatures (Xie et al.,

2014) and derive an RC model of the thermal coupling between the battery and the application processor (Xie et al., 2013). Their runtime policy minimizes the number of deadline misses for various synthetic real-time tasks by considering the thermal coupling between the battery and CPU. ARM's new *Intelligent Power Allocation (IPA)* (Muller, 2014) scheme aims to maximize performance under thermally limited scenarios by shifting the power between the heterogeneous CPU cores and GPU based on the expected performance return. Unlike the previous work, we do not attempt to improve performance under temperature constraints. Instead, we consider the target QoS levels as performance-wise sufficient and aim to sustain that QoS level for a maximum duration by optimizing transient state temperatures. Similar to IPA, Prakash et al. (Prakash et al., 2016) propose coordinated CPU-GPU DVFS to maximize FPS under a thermal constraint but consider CPU and GPU thermal couplings as well. Our work recognizes CPU-GPU thermal couplings to make thermally-efficient core allocation decisions.

For multi/many core systems, various work present thermally-efficient spatial allocation of threads (Khdr et al., 2015; Shafique et al., 2015; Khdr et al., 2017). In fact, the intuition behind thermal-coupling aware mapping in our QScale policy (Sahin and Coskun, 2016b) is similar to prior work. However, QScale 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.

Several studies address the scheduling of discrete DVFS states with thermal considerations in real-time systems domain. Applying faster switching between the discrete DVFS levels have been formally shown to maximize the workload under a thermal threshold (Chantem et al., 2009) and minimize the peak temperature (Chaturvedi et al., 2010) in hard real-time systems. Inspired by those techniques in real-time systems domain, we utilize DVFS scheduling to enable fine-grained CPU power tuning and meet the target QoS constraints with minimal use of the thermal headroom for improved performance sustainability.

3.1.2 Heterogeneity-aware Power and Thermal Management

Single-ISA heterogeneous architectures (e.g., ARM's big.LITTLE (ARM, 2013)) combine high-performance power hungry cores with simpler low-power cores to provide more energy-efficient operating points (Kumar et al., 2003; Kumar et al., 2004). Due to widely varying compute requirements across different mobile applications, such CPU architectures have been widely adopted in current mobile SoCs for improving energy efficiency. In this section, we provide an overview of runtime power and thermal management techniques proposed for such heterogeneous multicore CPUs and highlight the differentiating aspects of our thermal optimization solution for big.LITTLE (Sahin and Coskun, 2016b).

Scheduling and Energy Management on Heterogeneous CPUs: Some prior scheduling techniques maximize overall throughput on heterogeneous multi-cores running multi-program workloads. Koufaty et al. (Koufaty et al., 2010) dynamically monitor several hardware events to guide load balancing decisions in the Linux scheduler. Other work (Kumar et al., 2004) relies on application profiling on all core types to guide scheduling. On a big.LITTLE mobile platform, Hsiu et al. (Hsiu et al., 2016) 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. (Muthukaruppan et al., 2013) propose a real-life power budgeting framework on a big.LITTLE system where target QoS of multiple single-threaded self-adaptive applications (Hoffmann et al., 2010) are adjusted reactively. Various application-specific energy management policies have been proposed for heterogeneous mobile CPUs. Zhu et al. (Zhu and Reddi, 2013) analyze web-page features to determine DVFS settings while meeting latency constraints. Pathania et al. (Pathania et al., 2015) 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.

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. (Sharifi et al., 2010) propose a job allocation strategy for temperature balancing on a heterogeneous SoC to mitigate negative effects of thermal variations. Their technique relies on prior knowledge of power and performance characteristics of all applications and optimizes steady-state temperatures only. Kim et al. (Kim et al., 2015) 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. (Singla et al., 2015) provide a thermal modeling methodology using a real-life big.LITTLE platform and present a proactive DTM policy to prevent thermal violations. Our techniques, on the other hand, maximize the duration before a thermal violation occurs by providing thermally-efficient QoS management.

3.1.3 Application and QoS-Aware Runtime Management

Similar to our work in this thesis, several prior studies also consider the bursty and throughput-oriented characteristics of applications to tailor energy optimization policies. Hashemi et al. (Hashemi et al., 2015) investigate the bursty compute behavior in web applications for dynamic power management (DPM) 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.

(Rao et al., 2017) profile each mobile application offline to determine the performance sensitivity of an application to CPU and memory DVFS. They leverage this information at runtime to perform an application-specific control for minimizing energy under a performance target. Zhu et al. (Zhu et al., 2015a) propose a framework to distinguish bursty and throughput-oriented events in a web browser and allows to manage QoS and energy tradeoffs accordingly.

Various prior work has studied managing QoS tradeoffs with power and energy considerations. PowerDial (Hoffmann et al., 2011) automatically extracts application parameters to perform dynamic accuracy tradeoffs under a power cap. JouleGuard (Hoffmann, 2015) provides a learning based 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 harder to generalize to off-the-shelf mobile applications.

3.2 Capturing Non-deterministic Inputs for Repeatable Execution of Mobile Applications

Various existing tools aim at providing repeatable execution of mobile applications on the Android platform that is also our focus in this thesis due to its widespread adoption. We briefly review existing techniques to highlight the distinguishing aspects of our novel record and replay framework for Android, RandR.

Robotium (RobotiumTech, 2019) and Espresso (Espresso, 2019) allow developers to instrument their source code with test scripts that interact with the UI widgets of an application. These tools do not require modifications to the underlying Android framework and can provide cross-platform testing capabilities. However, they require significant manual effort to analyze the GUI layouts and write tests scripts to perform interactions with the app. As opposed to approaches that require manual efforts to derive test scripts, our goal is to provide a record and replay framework to capture natural user interactions with an application automatically.

Majority of the existing record and replay systems for Android (e.g., RERAN (Gomez et al., 2013), Valera (Hu et al., 2015), MobiPlay (Qin et al., 2016)) rely on raw screen coordinates of touch events or require privileged modes of operation (i.e., *root*). Such elevated privileges are challenging to achieve on off-the-shelf mobile devices (Qin et al., 2016). Since the specific coordinates of an interactive UI widget may also shift across different platforms due to varying screen size and resolutions, these tools are not suitable to achieve cross-platform replay. Besides, RERAN (Gomez et al., 2013) and MobiPlay (Qin et al., 2016) cannot reproduce execution variations due to non-deterministic inputs other than UI interactions (e.g., network events).

3.3 Understanding Malicious Non-deterministic Behavior in Mobile Applications

While different UI interactions or network inputs may intuitively alter the execution behavior of an application, an emerging class of evasive Android malware can also alter its behavior depending on the system the application is executed on (e.g., emulator or a real device). This section reviews prior work on discovering emulation detection methods and compares against our Proteus (Sahin et al., 2018) system. We also discuss existing defense approaches against evasive malware.

Finding Discrepancies of Emulation Environments: Jing et al. (Jing et al., 2014) identify a large number of detection heuristics based on the differences in file system entries and return values of Android API calls. For instance, presence of "`/proc/sys/net/ipv4/tcp_syncookies`" file or a `False` return value from the "`isTetheringSupported()`" API implies emulation. Such discrepancies can be easily concealed by editing Android's system images and API implementations to fake real device view (Liu et al., 2017; Bordoni et al., 2017). Petsas et al. detect QEMU-based emulation by observing the side effects of caching and scheduling on QEMU (Petsas et al., 2014). Other work leverages performance side channel due to low graphics performance on emulators to fingerprint emulation (Vidas and Christin, 2014). These techniques, however, have practical limitations as they require many repeated trials and observations which makes the malware easier to flag. Our work *systematically* uncovers observable differences in instruction semantics via large-scale analysis of detailed instruction-level program traces.

Similar to our approach, other works also aim at discovering discrepancies of emulators at instruction granularity. Various techniques (Martignoni et al., 2009; Paleari et al., 2009) execute randomized instructions on emulator and real hardware to identify the discrepancies of x86 emulators. To ensure coverage of a wide set of instructions, other work (Shi et al., 2014) carefully constructs tests cases with unique instructions based on manual analysis of the x86 ISA manual while our technique is fully automated. In addition, the analysis and findings of these studies are limited to x86 instruction set only while the vast majority of mobile devices are powered by ARM CPUs. In addition, these studies classify divergences based on instructions (e.g., using mnemonic, opcodes) which oversees the fact that even different instructions (e.g., `LDM` and `STM`) can diverge due to the same root cause (e.g., missing alignment check). Our study points to the unique root causes in the implementation of CPU emulators. Thus, our findings are readily useful for improving the fidelity of QEMU. Finally, as reliance on physical CPUs practically limits the number of test cases (e.g., instructions, register/memory operands, system register settings), we propose a novel scalable system which uses accurate functional models of ARM CPUs (i.e., Fast Models).

Martingoni et al. (Martignoni et al., 2012) used symbolic execution traces from a *high-fidelity emulator* to construct test cases that would achieve high coverage while testing a *low-fidelity emulator*. Lack of such high-fidelity emulator for ARM CPUs which power the vast majority of mobile devices, however, limits the applicability of this technique for our use.

Defense Against Evasive Malware: Several work proposes to detect divergent behavior in malware as a defense mechanism. Balzorotti et al. (Balzarotti et al., 2010) detect divergent behavior due to instruction semantics by replaying applications on emulators with the system call sequences gathered from real devices and comparing the runtime behavior. Lindorfer et al. (Lindorfer et al., 2011) propose a more generic methodology for detecing evasive malware based on the similarity of execution behaviors collected from a set of virtual machines. These approaches do not systematically expose potential causes of divergences that a future malware can use. Our work addresses the problem of proactively finding these instruction-level discrepancies and opens the possibility of pre-emptively fixing them.

Specifically for Android, other works (Liu et al., 2017; Bordoni et al., 2017) systematically remove observable differences from API calls, file system and properites of emulators and demonstrate resistance against evasion. Such approaches, however, require enumeration of root causes of discrepancies. Our Proteus system aids these approaches by enumerating the divergent cases between emulator and real CPUs.

3.4 Distinguishing Aspects of This Thesis

Power and thermal management are widely studied subjects for various computing systems from mobile to servers and data centers. Thus, we summarize the unique aspects of our thermally-efficient QoS management approach for sustainable performance as follows:

- Our work is first to show the performance drawbacks of existing thermal management approaches over extended durations (i.e., minutes or longer) due to the pursuit of favoring short term performance. Traditional thermal management has focused on maximizing performance under thermal limits (Bartolini et al., 2011; Singla et al., 2015; Prakash et al., 2016). Conversely, we make a case for *sustainable QoS* where our aim is to maintain a "just enough" level of performance for maximum durations. Our approach is inspired by inherent QoS tolerance of users (Zhu et al., 2015b), users' consistent QoS expectations over extended durations (e.g., as in gaming) (AOSP, 2019) as well as cooling restrictions in commodity mobile devices.

- We demonstrate, through real-life experiments, the viability of trading off short-term QoS for temperature reduction to extend the durations of sustained QoS.

- Our thermally-efficient QoS management solution, QScale (Sahin and Coskun, 2016b), brings the following novel contributions. First, we demonstrate the dependence of thermally-efficient core allocation decisions on dynamic CPU-GPU thermal couplings. Second, inspired by the studies demonstrating low TLP in mobile applications (Gao et al., 2014; Gao et al., 2015), we show that QoS is dominated by a few QoS-critical threads. QScale leverages this observation for thermally-efficient scheduling.

- Our application-aware QoS management framework, Maestro (Sahin et al., 2018), provides autonomous QoS-temperature tradeoffs for mobile applications. Maestro automatically reasons about the susceptibility of an application to thermal throttling. By detecting bursty and throughput-oriented compute behavior of real-world mobile applications at runtime, Maestro proactively manages QoS to increase durations of sustained performance.

- We implement and evaluate all of our techniques on real-life mobile platforms.

Table 3.1: Comparison of Android test and replay frameworks.

	Robotium (RobotiumTech, 2019)	Espresso (Espresso, 2019)	UI Automator (UIAutomator, 2019)	AppFlow (Hu et al., 2018)	RERAN (Gomez et al., 2013)	Valera (Hu et al., 2015)	Barista (Fazzini et al., 2017)	MobiPlay (Qin et al., 2016)	**RandR**
Automated record & replay	✗	✗	✗	✗	✓	✓	✗	✗	✓
No root privilege	✓	✓	✓	✓	✗	✗	✓	✓	✓
No custom OS	✓	✓	✓	✓	✓	✗	✓	✓	✓
Support closed-source application	✗	✗	✗	✗	✓	✓	✗	✓	✓
Replay randomized data	✗	✗	✗	✗	✗	✓	✗	✗	✓
Replay network data	✗	✗	✗	✗	✗	✓	✗	✗	✓
Cross-platform replay	✓	✓	✓	✓	✗	✗	✓	✗	✓

- In order to facilitate real-life studies with mobile applications, we propose a record and replay framework, RandR. RandR offers several unique advantages over prior work. First, RandR does not require any intrusive modifications to underlying OS/VM or root permissions to run. RandR achieves these practical capabilities through dynamic runtime instrumentation. Second, RandR associates UI events with their respective target UI widgets to provide coordinate-independent and cross-device replay. RandR can also record and replay non-deterministic input sources such as network events or random numbers beyond user inputs. Table 3.1 provides a comparison of RandR to existing work.

- Finally, we propose a system, Proteus, to identify the heuristics that could be leveraged by evasive Android malware at the instruction level. Proteus automatically identifies the instructions and conditions that cause divergence between emulated and real mobile devices. We identify several root causes behind a large number of discrepancies. We show that some of these root causes can be eliminated without any observable performance overhead.

Chapter 4

Runtime Management for Sustainable Performance

This chapter introduces our runtime management policies that incorporate insights from inherent QoS tolerance of users as well as from unique workload characteristics of mobile applications. Our policies aim at improving sustained performance under thermal constraints. In Section 4.1, we present a thermally-efficient runtime DVFS framework for reducing temperature under a given QoS constraint. Section 4.2 introduces our QScale framework for enabling efficient QoS-temperature tradeoffs on heterogeneous multicore mobile CPUs. Finally, in Section 4.3, we introduce a framework that leverages distinct computation patterns of mobile applications to autonomously guide QoS-temperature tradeoffs.

4.1 Thermally-Efficient DVFS under QoS Guarantees

DVFS is a widely employed technique for managing power/performance/temperature tradeoffs. While lower DVFS states can lower power and temperatures, reduced performance can hurt user experience. In this section, we present a thermally-efficient runtime DVFS technique for managing QoS levels. Our technique minimizes CPU temperatures while ensuring QoS guarantees required by the user and applications via formal closed-loop control. Section 4.1.1 and Section 4.1.2 presents our technique and evaluations on a real-life smartphone, respectively. Such a technique will allow users or system-level management policies to perform thermally-efficient QoS tradeoffs

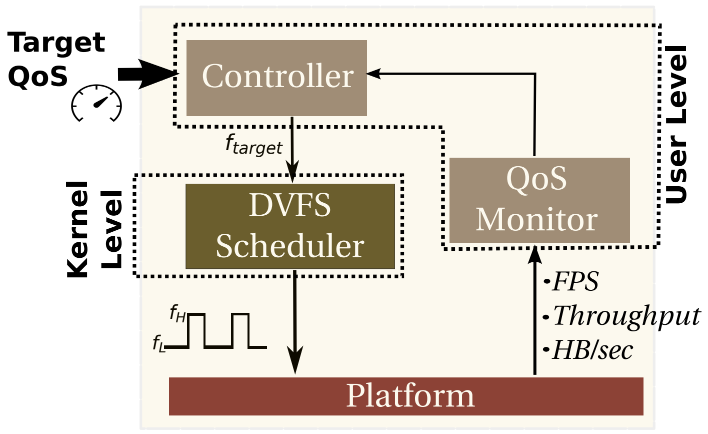


Figure 4·1: Overview of the implementation framework. Frames per second (FPS), throughput and heartbeat per second (HB/sec) correspond to the QoS metrics for our applications.

(e.g., based on battery level or thermal status).

4.1.1 Runtime Control Policy

This section introduces our runtime DVFS framework and policies for efficiently tuning the QoS to “barely” match the target levels in the pursuit of achieving longer durations of sustainable performance. Figure 4·1 presents an overview of our design which comprises of three main components. The *closed-loop controller* takes the runtime tunable QoS level as a performance target and determines the operating frequency of the processor. The *DVFS scheduler unit* converts the continuous target frequency provided by the controller into a time-scheduled sequence of available discrete DVFS levels to efficiently match the continuous target frequency with minimal thermal impact. The *QoS monitoring unit* periodically feeds the frames-per-second (FPS), throughput, or heartbeats/second (HB/sec) and into the controller.

Closed-loop QoS Controller

We design a feedback controller for dynamically adjusting the CPU speed to converge QoS towards desired levels. The controller tracks the progress of the application towards the target QoS by interacting with the *QoS Monitoring Unit* at every control interval. We use the following performance model, which represents the QoS level for the next time interval ($Q[k+1]$) as a fraction of the maximum achievable QoS (Q_{max}) at the highest frequency setting:

$$Q[k+1] = Q_{max}u[k] \tag{4.1}$$

$$e[k] = Q_{target} - Q[k] \tag{4.2}$$

The control signal $u[k]$ ranges between 0 to 1 and corresponds to the frequency scaling factor which determines the QoS level at time $k+1$. Since the goal of the controller is to minimize the difference between the target and current QoS levels, the error term $e[k]$ simply corresponds to this difference. The transfer function of the Equation 4.1 in the z-domain is given by:

$$F_1(z) = \frac{Q(z)}{U(z)} = \frac{Q_{max}}{z} \tag{4.3}$$

We find the transfer function $F_2(z)$ that defines the correspondence between the control signal and the error term by setting the following global transfer function of the closed-loop control system to $1/z$:

$$G(z) = \frac{F_1(z)F_2(z)}{1 + F_1(z)F_2(z)} \tag{4.4}$$

We obtain the discrete-time representation of the controller equation by substituting the $F_2(z)$ and taking the inverse z-transform as follows:

$$F_2(z) = \frac{U(z)}{E(z)} = \frac{z}{Q_{max}(z-1)} \tag{4.5}$$

$$u[k+1] = u[k] + e[k]/Q_{max} \tag{4.6}$$

We examine the stability and convergence of this control system by analyzing the global closed-loop transfer function $G(z)$ in z-domain. The closed-loop transfer function of $1/z$ has only one pole located at zero, which lies within the unit circle, ensuring the stabilization around the target QoS. Convergence to the target QoS level can be examined by evaluating the $G(z)$ at $z = 1$ and verifying a unit gain. Since $G(z) = 1/z$ evaluates to 1 at $z = 1$, the system has unit gain at the steady state and converges to the target QoS. The settling time of the controller is a function of the largest pole (a) of the closed-loop transfer function (Hellerstein et al., 2004), approximated by $-4/log(a)$. Since the $G(z)$ has its single pole located at 0, the controller can converge instantly, limited by the controller invocation period in practice.

Fine-grained DVFS State Scheduling

The controller provides a continuous output signal while the CPU can only support discrete DVFS levels. We propose a DVFS state scheduler which divides the controller period into bins and switches between two neighbor frequency levels to produce an average frequency that matches the controller output. We also demonstrate the potential to minimize the thermal impact by making thermally-aware scheduling decisions to further extend the durations of sustainable performance.

Minimizing the Thermal Impact with DVFS Scheduling: We use the following discretized version of a lumped RC thermal model similar to prior research (Skadron et al., 2002) to demonstrate the intuition behind our scheduling approach for minimizing the peak temperature of the CPU:

$$T[n+1] = T[n] + S(R_{th}P_k[n] - T[n])/R_{th}C_{th} \tag{4.7}$$

$$T[n+1] = c_1 T[n] + c_2 \hat{T}_k \tag{4.8}$$

where S is the sampling period, R_{th} and C_{th} are the thermal resistance and capacitance, $T[n]$ is the temperature at sampling interval n, $P_k[n]$ is the power level corresponding to DVFS state k, and $\hat{T}_k = R_{th}P_k[n]$ is the steady-state temperature for the power level P_k.

Consider the case where the scheduler estimates M bins to be scheduled with the higher frequency state in the following control interval, and those high frequency states are applied at distances of L. Next, we show that scheduler can reduce the peak temperature by increasing the distance L. Using Equation 4.8, we write the peak temperature at the end of the M^{th} high frequency state as follows:

$$T_p = c_1^{ML}T[0] + c_2 \sum_{i=0}^{M-1} c_1^{iL}\hat{T}_h + c_2 \sum_{i=0}^{(M-1)} \sum_{j=iL+1}^{(iL+L+1)} c_1^j \hat{T}_l \tag{4.9}$$

Since c_1 and c_2 are less than zero, when the distance between the high frequency states (L) is increased, the last term dominates and temperature approaches lower steady state temperature $\hat{T}_l$. Thus, distributing the high frequency states furthest from each other reduces the increase in temperature. Based on this intuitive observation, our scheduler implements maximum spatial distribution of the high frequency state bins within the control interval.

Impact of DVFS Granularity on Temperature and Performance: As a result of thermal time constants, temperature exhibits a "gradual" increase or decrease than a step thermal response. Thus, in addition to efficient DVFS scheduling, applying the DVFS state decisions faster, or increasing the number of switches within the interval, can also reduce maximum temperature. This is due to the thermal buffer provided by the thermal time constants (Chantem et al., 2009). One critical issue that limits the thermal-optimization via quick DVFS on a real hardware is the performance overhead of switching. This case is pointed out in Figure 4·2 which shows the measured performance overhead over different switching granularities. Reducing the

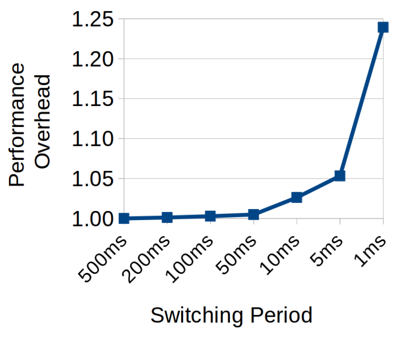


Figure 4·2: Performance overhead of DVFS.

switching frequency beyond 5ms can incur as much as 25% performance overhead.[5] Based on this analysis, we set our fine-grained DVFS period to be 20ms.

4.1.2 Evaluation

This section describes the details of our experimental evaluation methodology as well as presentation and discussion of our results.

Implementation and Experimental Setup

Hardware Platform: Our target experimental platform is a state-of-the-art Qualcomm Snapdragon MSM8974 smartphone (Snapdragon MSM8974 MDP., 2014) that hosts a Snapdragon 800 SoC (used in many modern smartphones, e.g., Nexus 5 and Galaxy S4). The Snadragon 800 SoC consists of a Quad Core Krait 400 CPU along with an Adreno 330 GPU, 2GB LPDDR3 RAM and is powered by a 1,600mAh Li-ion battery. The phone runs Android KitKat version 4.4.2 and Linux 3.4.0 kernel. The Krait 400 CPU supports 12 operating frequencies ranging from 300MHz to 2.1GHz. Temperature measurements can be done on a per-core basis via on-chip thermal sensors. Sensor readings for the CPU cores, battery and skin temperature are performed using the thermal virtual file system provided by the Linux kernel (i.e., /sys/class/thermal) with ±1°C accuracy. We use the *logcat* system debugging tool available as part of the Android framework for monitoring the frames per second and use *perf_event* kernel API for accessing hardware performance counters. Our phone allows for measuring only the overall power consumption using the voltage and current sensors. For the CPU applications that do not require graphical interface, we turn-off the LCD display throughout the measurements. We leave the LCD display on for the GPU applications.

[5]This experiment is performed on state-of-the-art Qualcomm MDP8974 smartphone.

Application	**Category**	**QoS Metric**
Sjeng	Artificial Intelligence	Throughput
H.264	Media Processing	Throughput
LU	Math	Throughput
Pearl Boy	Graphics/WebGL	Frames per second
Aquarium	Graphics/WebGL	Frames per second
Bodytrack	Computer Vision	Heartbeats/sec

Table 4.1: Summary of applications and respective QoS metrics.

Application Set: Mobile systems run a broad range of applications and a single performance metric cannot gauge performance of all applications. Thus, we construct a benchmark set for our experiments by combining applications from various domains and evaluate them using different QoS metrics as summarized in Table 4.1. The LU application, a common kernel in many image/video processing and mobile healthcare applications, is selected from Scimark 2.0 (Pozo and Miller, 2000), which is a benchmark suite for testing Java based platforms. A video encoding (H.264) and an artificial intelligence application (Sjeng) are chosen from the SPEC CPU2006 (Henning, 2006). We use two online graphics applications created with WebGL, Aquarium (Aquarium, 2018) and Pearl Boy (Goo, 2018). The Aquarium shows an animation of fishes in a tank, while the Pearl Boy is an interactive application that requires directing a boat in the sea. To ensure consistency between the runs, we automate the user interaction by applying the same sequence of *input swipe* commands for each experiment through a lightweight background shell program. We also use Heartbeats (Hoffmann et al., 2010) instrumented version of the bodytrack computer vision application from the PARSEC suite (Bienia et al., 2008). Heartbeat framework allows to monitor application-specific QoS using a standardized interface and, for the bodytrack application, this framework emits a heartbeat whenever the processing of one scene is completed. Since the Heartbeats framework can be applied to a wider domain of applications for QoS monitoring and tuning purposes, we find value in showing the applicability of our techniques on a Heartbeat-instrumented application.

Baseline Policies: The default CPU frequency scaling policy in our phone (and in most state-of-the-art Android devices) is the *ondemand governor* (Brodowski, 2012), which adjusts the CPU frequency based on the CPU load. Thus, we use the ondemand governor as our baseline power management scheme in our experiments.

Thermal throttling policies operate hierarchically with the CPU frequency governors and assign maximum frequency limits for ensuring operation below a thermal set-point. The CPU governors cannot use the frequencies that are above the assigned limit. Since the control-theoretic thermal management solutions are among the most commonly used techniques for maintaining the maximum temperature at a given threshold, we use a DVFS-based PID controller as the baseline CPU throttling mechanism. Modern smartphones also incorporate skin temperature management policies to keep the outer device temperature within the human comfort levels. Thus, performance degradations can occur due to increased skin temperatures as well. Since our MSM8974 device does not provide a skin temperature management policy by default, we implement the skin thermal management scheme available in the Nexus 5 smartphones. This policy assigns a maximum CPU frequency limit whenever a skin temperature trip point is reached, as described in Table 4.2. Both throttling mechanisms poll the thermal sensors and assign frequency limits every 100ms. We choose 100ms as it provides non-intrusive (less than %1 execution overhead) thermal management while maintaining sufficient time granularity to avoid thermal emergencies.

Runtime Implementation: We implement the closed-loop controller as a user-level program that regularly monitors the QoS level and passes the target frequency to the

Trip Point	Frequency Limit
40°C	1.9GHz
42°C	1.5GHz
44°C	1.2GHz

Table 4.2: Temperature thresholds and target frequency limits of the baseline skin temperature controller.

kernel-level DVFS scheduler. The controller is invoked every 200ms for the CPU applications and every 1 second for the GPU applications. We have observed noise in the FPS values when sampling at a finer granularity. We implement our DVFS scheduler in the kernel level as a new CPU governor with a *sysfs* interface to allow for assigning target frequency levels from the user space. The governor based implementation allows users to easily enable/disable our QoS tuning policy. The DVFS scheduler applies the frequency decisions at the granularity of 20 miliseconds via the *cpufreq* (Brodowski, 2012) interface. We have measured the frequency transition latency in our system to be 186.4 microseconds by wrapping the *cpufreq_driver_target* call in our kernel module with timing utilities. We have found 20 miliseconds to be the finest DVFS granularity that could be applied without introducing noticeable overhead (<1%) in our system. The maximum performance overhead of our framework is less than 1.3% across all the applications in our benchmark set.

Results and Discussion

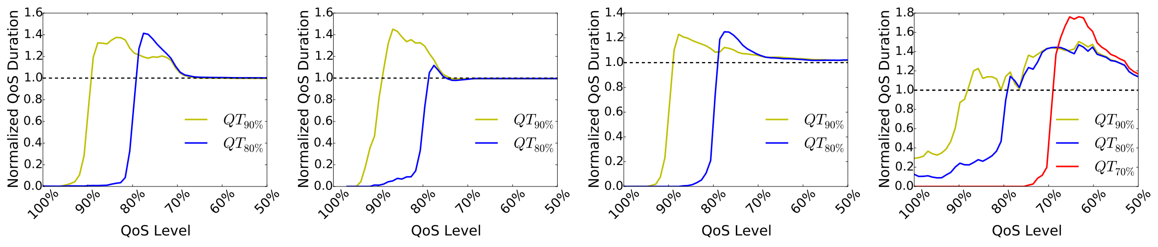
In this section, we present a thorough evaluation of the thermally-efficient QoS tuning policy that we have described and demonstrate its benefits for achieving longer durations of sustained performance. We evaluate the CPU applications with the CPU temperature triggered dynamic thermal management policy (DTM_{cpu}), which is a PID controller based throttling scheme described previously in the section. For the graphics applications, we have observed that CPU temperatures did not reach to critical limits while the skin temperatures kept increasing over time. Thus, we also provide an evaluation of our QoS tuning policy with the skin temperature controller (DTM_{skin}) running as the throttling mechanism on our platform. We aim to show the benefits of our QoS tuning approach from the performance sustainability perspective under both processor and device-level skin temperature constraints.

		H264			Bodytrack			Sjeng			LU			
		ondemand & DTM	QT90	QT80	ondemand & DTM	QT90	QT80	ondemand & DTM	QT90	QT80	ondemand & DTM	QT90	QT80	QT70
Before Throttling	Average QoS	0.99	0.89	0.80	0.96	0.902	0.804	1.01	0.896	0.805	0.995	0.903	0.806	0.707
	Standard Deviation of QoS	0.035	0.042	0.037	0.02	0.028	0.043	0.086	0.059	0.065	0.107	0.109	0.075	0.074
Overall Execution	Average QoS	0.85	0.85	0.79	0.871	0.872	0.803	0.85	0.84	0.79	0.723	0.756	0.732	0.695
	QoS Degradation	27.2%	16.6%	**0.3%**	18.3%	11.3%	**0.1%**	28%	18%	4%	35%	30%	22%	7.4%
	Time Spent in Throttling	85.4%	55.5%	**6.8%**	59.4%	48.3%	**0%**	86.9%	61.7%	26.1%	87.7%	86.6%	73.9%	**45.4%**
	Average Power	0.99	0.97	0.88	0.92	0.97	0.86	0.99	0.96	0.89	0.97	1.02	0.97	0.91
	Energy Consumption	1.01	0.98	0.94	0.91	0.97	0.93	1.01	0.98	0.96	1.02	1.01	0.99	0.97
	QoS/Watt	0.99	1.02	1.05	1.09	1.04	1.08	0.99	1.013	1.03	0.997	0.994	1.006	1.03

Table 4.3: A summary of results for the CPU applications. QT(X) represents proposed QoS tuning policy with X% target QoS. Average QoS, power, energy and QoS/Watt values are normalized to the highest static frequency setting (2.1GHz). QoS degradation corresponds to the percentage of QoS loss from the first to the last iteration of the run.

Performance Sustainability Under CPU Temperature Constraints: For the CPU applications, we explore three target QoS levels which are set to 90%, 80% and 70% of the average QoS achieved when the application is run at the highest static frequency setting on an initially cold system. For clarity, we do not present results for the 70% cases if the application QoS does not degrade to this level using the highest static frequency setting. We emulate extended application durations by repetitively running the applications for a fixed number of iterations. We determine the number of iterations based on a maximum battery temperature limit of 50 °C.

Table 4.3 gives a detailed overview of our experimental results where QT(X) corresponds to the proposed QoS tuning technique at the target QoS level of X%. All the values except for the QoS degradation and the time spent in throttling are normalized



(a) H264 **(b)** Bodytrack **(c)** Sjeng **(d)** LU

Figure 4·3: Normalized duration of time spent above a QoS level by the proposed QoS tuning policy for different target QoS level. "$\text{QT}_{\text{X\%}}$" represents the proposed QoS tuning policy with X% QoS goal. A data point in the figure corresponds to (Time spent above a QoS with QT)/(Time spent above a QoS with (DTM_{cpu}+ondemand).

to the highest static frequency setting. We evaluate the phases of the execution without throttling (indicated by "before throttling" in Table 4.3) separately to examine the controller's ability to meet QoS goals without the interference of the throttling policy. Overall, our controller is able to effectively meet the given QoS targets with less than 0.06 average deviation. In most cases, average QoS of the overall execution is lower than the "before throttling" phase due to the performance impact of the throttling. However, bodytrack and h264 applications are able to sustain the performance close to 80% QoS level throughout the whole execution as little or no thermal throttling is incurred at that level for these two applications. The highest QoS degradation is observed for the LU application, which spends 45% of time in throttling even in the lowest QoS target of 70%. This degradation is due to the power hungry nature of this CPU intensive computing kernel that quickly reaches the CPU thermal limits. For all benchmarks, the baseline *ondemand* policy continuously seeks to convert the thermal headroom into performance by scaling the frequency to high levels. This incurs the highest QoS degradation due to the increased percentage of time spent in throttling. The QoS tuning policy with 90% target level achieves 38.6% reduction in throttling duration on average and consistently provides lower QoS degradation for all benchmarks. The QoS tuning also provides up to 14% and 7% reductions in power and energy, respectively.

Figure 4·3 presents the improvements in performance sustainability for our CPU applications. The figure shows the duration of time spent above a QoS level with the proposed QoS tuning (QT) policy as normalized to the baseline. The proposed technique provides substantially longer execution time around the given QoS target. This could be observed in Figure 4·3a,4·3b and 4·3c where the the curves start to rise significantly above the dashed line (normalized baseline) when approaching the the given QoS goals. For the h264, bodytrack and sjeng applcations, an average of 37% and 26.7% longer sustainability is achieved for the 90% and 80% QoS levels, respectively. Improvement by the QoS tuning on the bodytrack application for the 80% QoS level is lower (11%) as the QoS drops to 80% range for only a short duration of time with the baseline policy. The LU application, as shown in Figure 4·3d, provides the peak improvements in sustainability with 74% longer duration the 70% QoS target is sustained. This application has the highest power consumption among our applications and quickly reaches to thermal limits with the baseline setting. Therefore, using higher frequency settings results in higher QoS degradation for this application. In fact, the proposed policy is unable to sustain the QoS around the target range for the higher 90% and 80% target levels and QoS distribution shifts towards a lower range.

Fine-grained QoS Control with DVFS scheduler: In addition to providing fine-grained QoS control, the DVFS state scheduler also aims to achieve the maximum spatial distribution of the higher frequency states to minimize thermal impact. Figure 4·4 shows the effect of such distributed scheme on the temperature trace of the h264 application. Undistributed scheme simply switches from low frequency state to high frequency state only once during the control period. The distributed policy applies finer granularity switching with maximum possible low frequency periods between

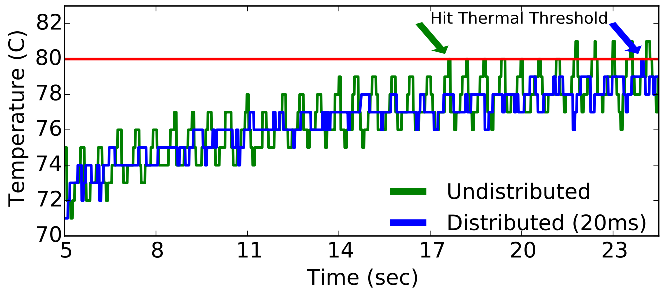
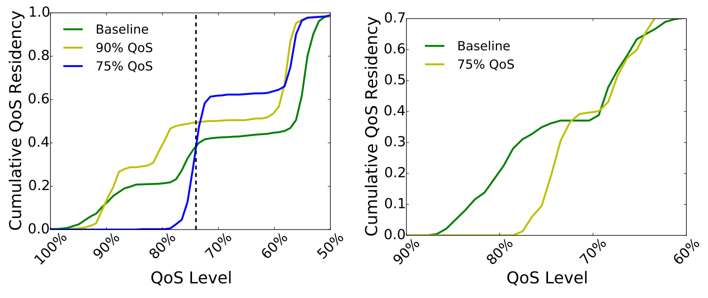


Figure 4·4: Temperature traces for two DVFS scheduling schemes. Undistributed scheme switches from lower to higher frequency only once during the control interval (1 sec). Distributed scheme gains more thermal headroom by applying fine-grained DVFS and scheduling high states farthest possible from each other. The duty-cycle is 33% high.

the high frequency states. Both policies provide the same average frequency. As annotated by the two arrows in Figure 4·4, the distributed policy allows for longer execution without reaching to the thermal limit.

Performance Sustainability Under Skin Temperature Constraints: We further investigate the applicability of our motivation and QoS tuning technique under skin temperature constraints for extending the durations of target FPS levels. Figure 4·5 shows the cumulative distribution of the QoS for both applications during a 15 minutes of continuous execution. 100% QoS corresponds to 40 FPS for the Aquarium application and 60 FPS for the Pearl Boy application. A data point corresponds to the fraction of the overall execution time spent above the corresponding QoS level. For the Pearl Boy application, QoS tuning with 75% target level improves the sustainability by 9%, from 36% to 40% of the execution time spent above the target. We do not show the 90% case as the baseline policy provides a QoS range below 88%. Figure 4·5a shows the cumulative QoS distribution for the Aquarium application and the dashed line corresponds to the 30 FPS limit which is pointed by prior research to be the lowest frame rate in the user tolerable range (Pathania et al., 2014)(Zhu et al., 2015b). The QoS tuning policy with 75% target (30 FPS) increases the sus-



(a) Aquarium **(b)** Pearl Boy

Figure 4·5: Cumulative QoS distribution for the two WebGL graphics applications. Dashed line in the left figure shows the 30FPS limit. The baseline policy corresponds to ondemand+DTM$_{\text{skin}}$.

tainability of this QoS level from 40% of the execution time to 62%, *providing 55% longer duration that the user can be provided with an acceptable FPS level.*

4.2 Enabling Efficient QoS-Temperature Tradeoffs on Heterogeneous CPUs

Single-ISA heterogeneous multi-core processors (Kumar et al., 2004) (e.g., ARM big.LITTLE (ARM, 2013)) 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 (Pathania et al., 2015; Seo et al., 2015; Zhu and Reddi, 2013). While running computationally demanding applications, current power management and scheduling techniques for big.LITTLE greedily maximize quality-of-service (QoS) within thermal constraints using power-hungry cores, leading to severe QoS loss over time. To provide mobile users with *sustainable QoS* over extended durations, we present *QScale*, a framework to minimize heat generation while precisely delivering desired QoS levels. Section 4.2.1 describes the details of QScale design and Section 4.2.2 provides an evaluation of our results obtained through a real-life implementation of QScale.

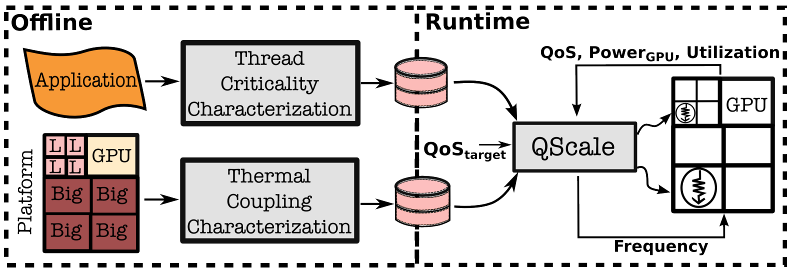

Offline
Runtime
Application
Thread Criticality Characterization
Platform
L L L L
GPU
Big Big Big Big
Thermal Coupling Characterization
QoS, Power_GPU, Utilization
QoS_target
QScale
GPU
Frequency

Figure 4·6: An overview of the proposed framework.

4.2.1 Proposed QScale Framework

This section describes QScale, a novel *thermally-efficient QoS* management framework for heterogeneous mobile platforms. Figure 4·6 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. We also identify the threads of an application that are critical to user-experience during the offline phase. QScale's runtime component 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 policy also performs closed-loop DVFS control to precisely meet the desired QoS.

Thermal Coupling Characterization and Awareness

GPU-CPU Thermal Coupling: First, we demonstrate the thermal coupling effect between the GPU and the big cores by running our GPU microbenchmark to stress GPU with as much isolation from the CPU as possible. In this experiment, the GPU operates at peak utilization and at the highest frequency for 1 minute. Figure 4·7 shows the resulting temperature increase (from an initially cold system) and the maximum temperatures of the GPU and CPU cores. As

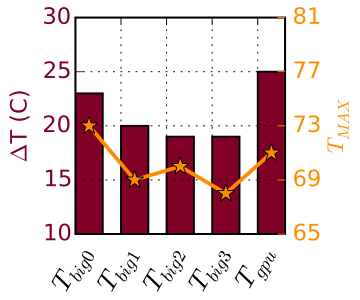


Figure 4·7: GPU thermal coupling in Exynos 5422.

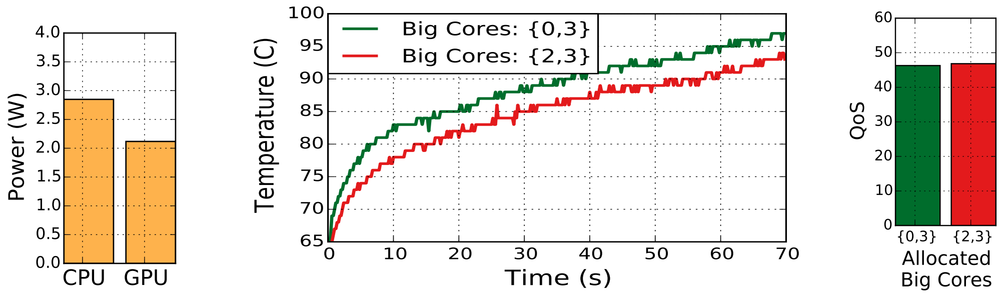
the GPU temperature increases from 46°C to 71°C by 25°C, we measure significant heating on all idle big cores. $Core0$ suffers the most from thermal coupling and its temperature increases by 23°C. Cores heat up at different rates due to their different locations on chip and varying proximities to the GPU.

Need for Coupling-aware Core Allocation: Next, we demonstrate that the impact of GPU-CPU thermal coupling is *application-dependent*. Figure 4·8 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[6] ($\{0, 3\}$ 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 $Core0$ 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.

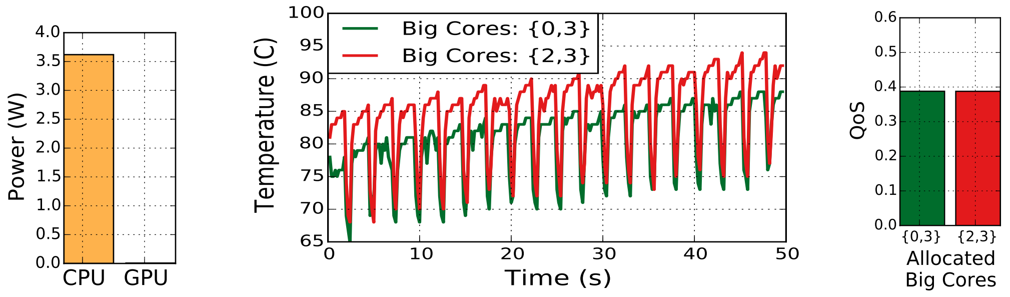
Criticality-Driven Scheduling for big.LITTLE

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 (Chung, 2012). 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

[6]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 application.



(b) Bodytrack application.

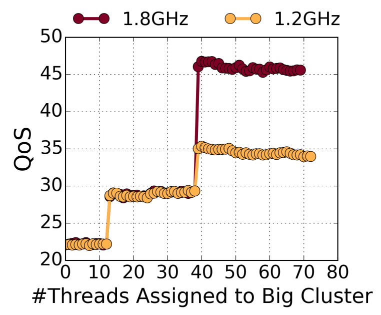
Figure 4·8: Power breakdown (left), temperature (middle) and QoS (right) for aquarium and bodytrack under different core assignments ({0,3} and {2,3})

cores). This observation is in line with the recent work (Gao et al., 2015; Seo et al., 2015), 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.

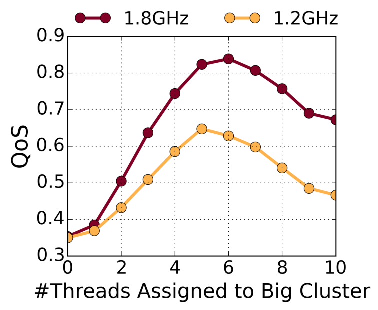
Offline Characterization: The aim of this offline characterization step is to identify the QoS-critical threads of an application that provide the highest QoS gains when allocated on a big core. We perform this characterization on each of our 6 applications. First, we allocate all the application threads 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 4·9 shows the QoS scaling and Figure 4·10 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 (Bienia et al., 2008), 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 app 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.

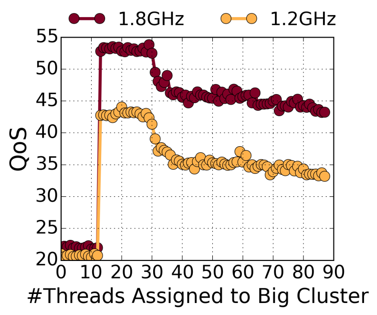
While our approach requires offline thread profiling for every application, we argue that such approach is profitable specifically for mobile applications for various



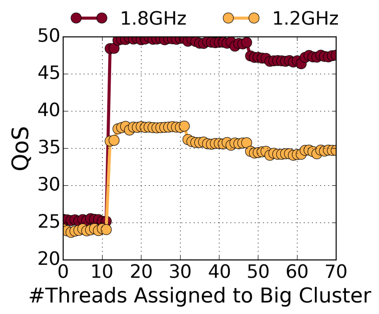
(a) Aquarium



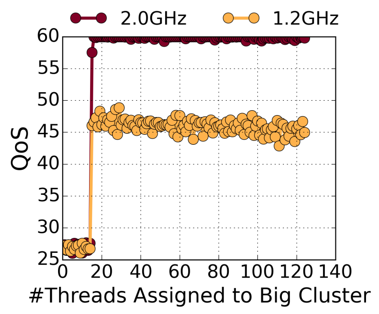
(b) Bodytrack



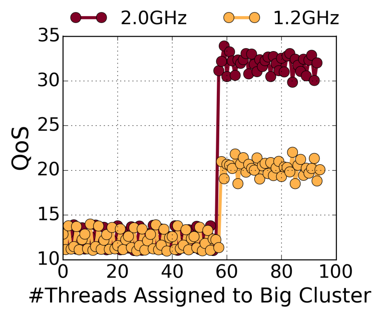
(c) EoT



(d) Rain



(e) Racing



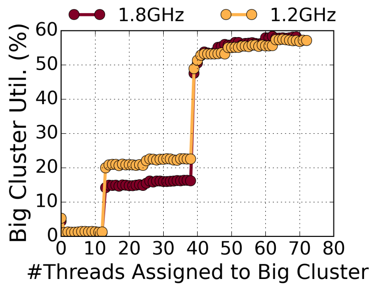
(f) RP

Figure 4·9: QoS scaling achieved by moving individual application threads from the little to the big cluster.

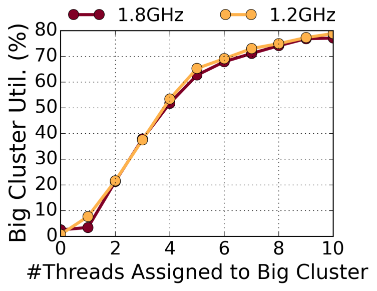
reasons: (1) Users will likely run the same application many times in the device life-cycle; (2) such offline profiling of applications can be automatically performed on a device without user interference (i.e., while device is left in charging) using Android record/replay tools (Gomez et al., 2013; Hu et al., 2015) as we have done in this work using RERAN (Gomez et al., 2013).

Online QoS Control Policy

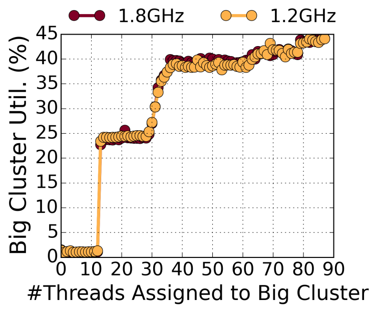
Overview: The goal of QScale's runtime policy is to deliver desired QoS levels while minimizing temperature by coordinating thermally-efficient scheduling with DVFS. It monitors the GPU power dissipation to select the most thermally-efficient set of big cores for executing the QoS-critical threads of an application. The policy also performs control-theoretic DVFS to precisely meet QoS targets. Target QoS can be



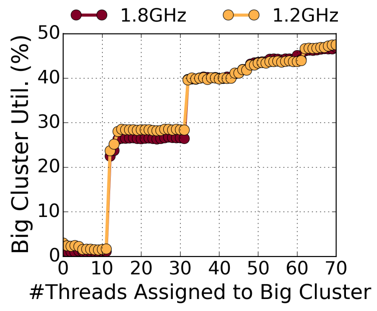
(a) Aquarium



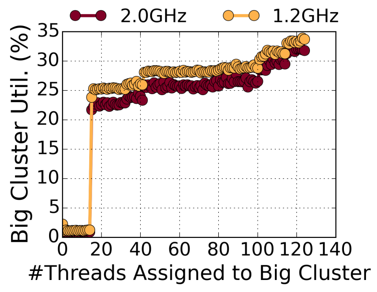
(b) Bodytrack



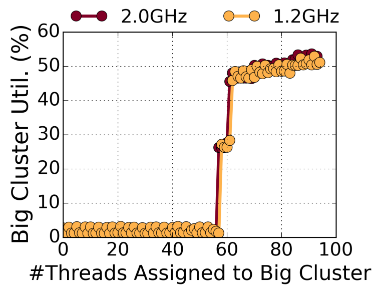
(c) EoT



(d) Rain



(e) Racing



(f) RP

Figure 4·10: Increase in big cluster usage as individual application threads are moved from the little to the big cluster.

dynamically adjusted upon user request, autonomously set by the system-level policies (e.g., based on battery level or thermal status), or statically set to minimum levels based on the limitations of user perception (Endo et al., 1996; Zhu et al., 2015b).

The top-level runtime control algorithm of QScale initially assigns the threads to the little cluster and, at every second, invokes our mapping policy 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 4·11). 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.

In case of thread-to-core mappings, if the number of critical threads is less than 4, there exists opportunity to lower temperature by making thermally-aware mapping.

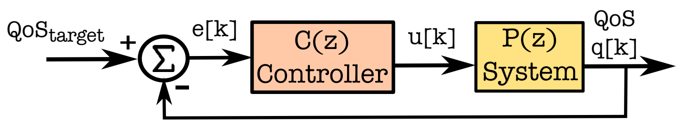


Figure 4·11: Feedback-based performance state control.

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 4·12). We collectively refer to DVFS and M-states as performance states. Migration states perform frequent (every 100ms) switching between the `Little`$_{1.4GHz}$ and `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 perceived latency. We do not migrate more frequently than every 100ms to avoid migration overhead (Kim et al., 2015; Muthukaruppan et al., 2013). 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.

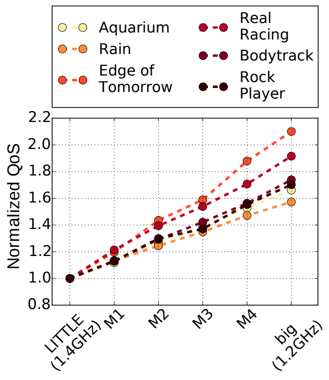


Figure 4·12: M-States.

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 4·11) 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 4.10. The pole (p) of global transfer function should be in range $[0, 1)$ to ensure stability and avoid oscillatory behaviour (Hellerstein et al., 2004). The value of p also allows to tradeoff robustness for responsiveness (Hellerstein et al., 2004) 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 ($G(z = 1) = 1$).

$$u[k] = u[k-1] + \frac{e[k](1-p)}{Q_{max}} \tag{4.10}$$

4.2.2 Evaluation

This section presents our experimental testbed, data monitoring/collection methodology and application set along with the implementation details of QScale. We also provide a comprehensive evaluation of QScale through experiments on a state-of-the-art big.LITTLE development platform.

Implementation and Hardware Setup

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

Table 4.4: 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

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 (ARM, 2013) 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 (ARM, 2013). 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.

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 4.4 provides a list of applications that we use in our experiments. We run the bodytrack computer vision application from the PARSEC suite (Bienia et al., 2008) where the frame-rate (or heartbeat-s/sec) is dynamically monitored by instrumenting the application with the

Heartbeats framework (Hoffmann et al., 2010). 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 (Gomez et al., 2013), to automate the execution by injecting a pre-recorded set of GUI events. Rain (Sheepeuh, 2018) and Aquarium (Aquarium, 2018) are web-based online animations that we execute within the Chrome web browser. All of our applications are multi-threaded. Figure 4·13 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.

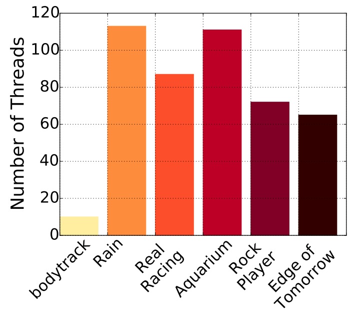


Figure 4·13: Number of threads.

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.

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* (Brodowski, 2012), 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 (Chung, 2012) 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.

Results and Discussions

This section evaluates QScale's effectiveness for maximizing durations of target QoS levels. In addition to the default *Interactive governor* (Brodowski, 2012) and *HMP scheduler* (Chung, 2012) pair, we also compare QScale to a *DVFS-only* policy. *DVFS-only* policy performs closed-loop DVFS to meet target QoS and uses the default HMP scheduling framework as opposed to the our 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 at the default management setting. 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* 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

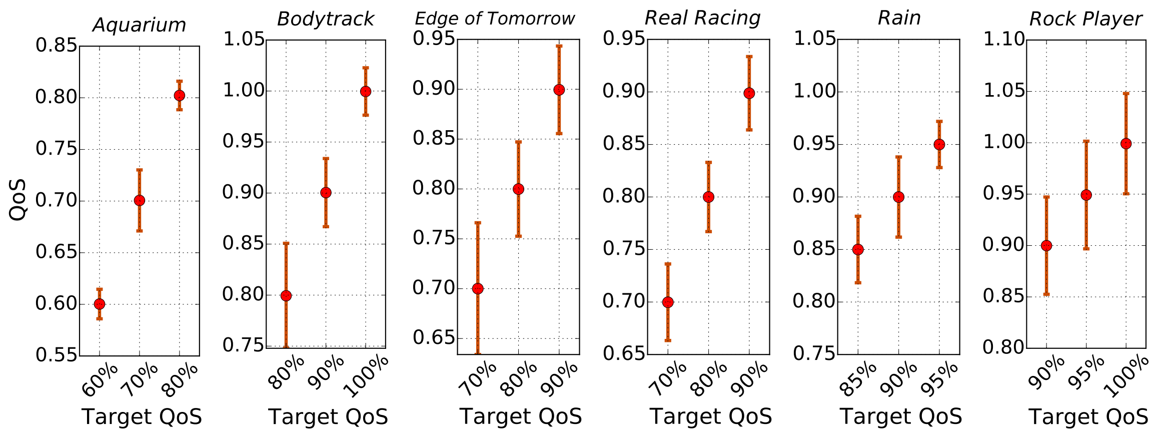


Figure 4·14: Average and standard deviation of QoS when using QScale under different target QoS levels.

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 4·14 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 4·15 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

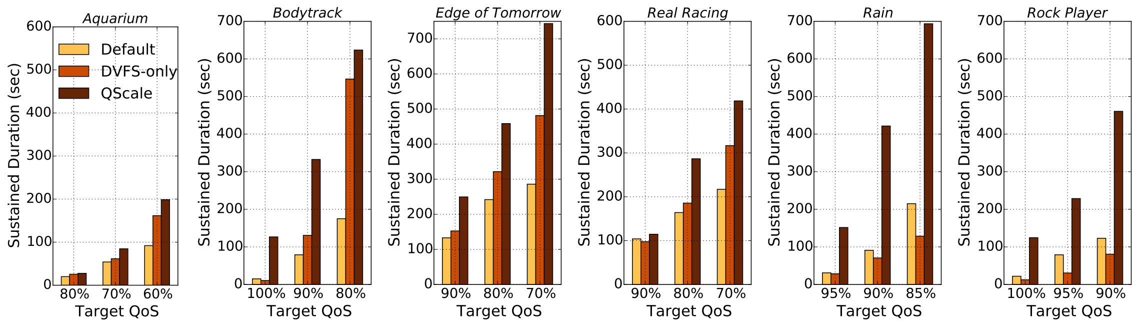


Figure 4·15: Sustained QoS durations with default management (*Interactive Governor + HMP scheduler*), *DVFS-only* and QScale policies under different QoS targets.

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 4·16, we show QoS and power at different big cluster DVFS levels when HMP scheduler or criticality-aware mapping is enabled. By moving the threads that bring the most QoS gains onto big cluster, QScale delivers the maximum QoS levels achieved using the HMP scheduler

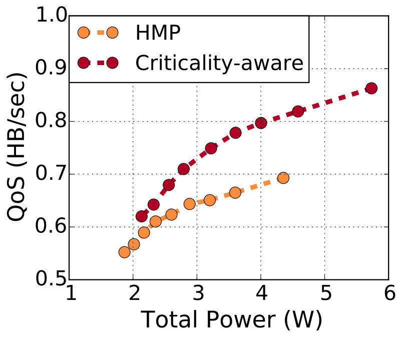


Figure 4·16: QoS/power scaling for *bodytrack*.

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 4·9 and Figure 4·10. 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.

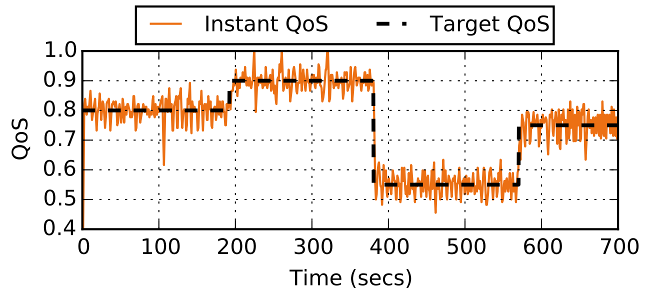


Figure 4·17: Adapting to dynamic QoS targets with QScale while running the *Edge of Tomorrow* gaming application.

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 4·17 presents the QoS trace from this experiment and shows that QScale closely tracks target QoS levels.

4.3 Autonomous QoS Management for Mobile Applications

While the runtime techniques presented in Sections 4.1 and 4.2 can provide thermally-efficient QoS management, they cannot autonomously manage the QoS decisions. In this section, we address the problem of autonomously guiding QoS decisions to improve sustained performance under thermal constraints. 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 (Sahin and Coskun, 2016b; Sahin et al., 2015) or seek to achieve sustained performance with low-power modes in a QoS- and application-agnostic manner, which can result in unacceptably low QoS (e.g., Android's Sustained Performance API).

This section describes our proposed Maestro framework for achieving autonomous and application-aware QoS tradeofss. Maestro proactively trades off QoS for the applications that are prone to large thermally-induced QoS loss and leverages our QScale policy to enforce the target QoS decisions. In the following two sections, we describe the details of our Maestro framework and present a real-life evaluation.

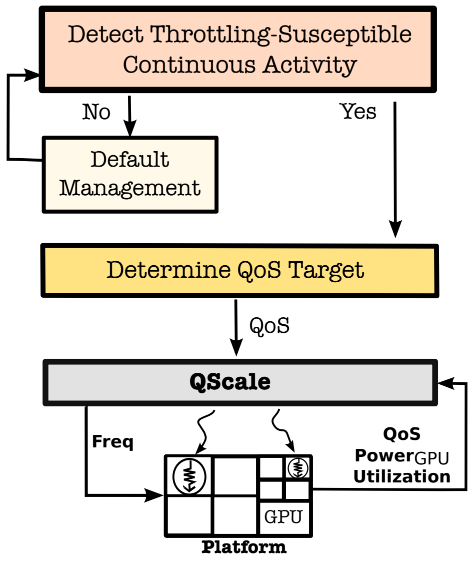


Figure 4·18: Overview of Maestro.

4.3.1 Maestro Framework

Our Insight: Maestro builds upon a novel insight on the relation between the computation characteristics of mobile applications (i.e., bursty vs. throughput-oriented) and the QoS impact of thermal throttling. While throughput-oriented mobile applications (e.g., gaming, video processing) may suffer from long-term throttling due to continuous computations and require QoS tradeoffs to sustain an acceptable throughput (e.g., frames-per-second, FPS), some applications only generate short bursts of computations in response to user interactions. Web browsing and many interactive Android applications (e.g., news reading, social networking, messaging, document reading) are examples of such bursty applications, where latency of the computations is the main factor that impacts the user perceived QoS (Endo et al., 1996). 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 distinguishes throttling-susceptible continuous computations from latency-sensitive bursty tasks, and manages QoS accordingly.

Overview: Figure 4·18 gives an overview of our proposed technique that is comprised of 3 main components. The online detection policy 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. Upon detecting such a computation phase, Meastro switches to using QScale for sustainable performance instead of the default Android management. Depending on how much the application is likely to suffer from throttling, Meastro scales the target QoS accordingly and provides as an input to QScale. QScale (Section 4.2) 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.

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. Such detection allows us to take proactive actions before the system heats up aggressively over time.

We devise a simple yet effective online policy (Figure 4·19) 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

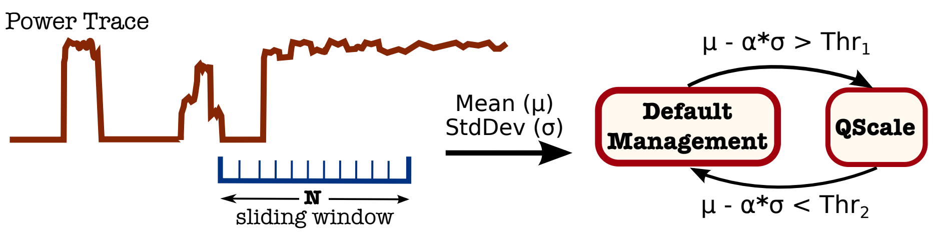


Figure 4·19: Sliding window based online detection policy.

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 Figure 2·4). We track mean (μ) and standard deviation (σ) using a 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).

Tuning Policy Parameters: Since the behavior of the proposed policy would depend on its parameters (i.e, N, Thr_1, Thr_2, 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 seconds window size as vast majority of bursty activities finish within 10 seconds (Zhang et al., 2013; Zhu et al., 2015a). 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.

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 (Yan et al., 2016) 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 the previous section. This mechanism is illustrated in Figure 4·20. Our intuitive rationale is that, as the power increases, choosing high QoS settings will quickly exhaust the thermal 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 minutes of duration without throttling for a typical possible high power application (i.e., >3.3W by *Bodytrack* in our example application set). We simply proportionally scale QoS within the 2W-3.3W range by choosing a 10% lower target QoS rate for every 600 mW as illustrated in Figure 4·20. Aiming for longer/shorter than 2 minutes minimum target sustained duration translates into larger/smaller steps in the tradeoff, approaching/distancing from the undesirable QoS region illustrated in Figure 4·20. We considered 30 FPS and 24 FPS as the minimum desirable QoS for 3D graphics and video playing scenarios, respectively.

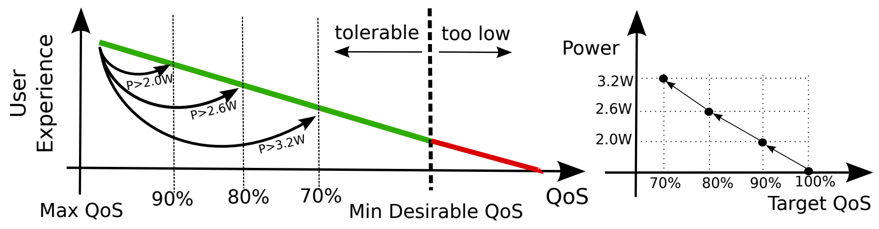


Figure 4·20: QoS tradeoff and determining QoS targets for QScale.

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 upwards. 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.

4.3.2 Experimental Setup

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

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 (ARM, 2013) 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 8-stage in-order pipeline (ARM, 2013). The A15 core supports 9 frequency levels from 1.2 GHz to 2 GHz while the

Table 4.5: 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 (CamanJS, 2018)	Apply different filters on an image	Latency
Bodytrack	Process image files	Heartbeats/sec
Edge of Tomorrow	Loading, menu selection and gaming	FPS
Aquarium (Aquarium, 2018)	Watch online animation	FPS
Rain (Sheepeuh, 2018)	Watch online animation	FPS
Rock Player	Open and play a video file	FPS

A7 core operates on 5 frequency levels between 1 GHz 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 6 frequency levels ranging from 177 MHz 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 of Android 7 for evaluation due to its direct relevance to our work.

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 4 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. Frames-per-second (FPS) is measured by querying the logs generated by the *Surface-Flinger* 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.

Applications: We braodly classify mobile applications into two classes as latency-sensitive bursty (e.g., browsing, interactive applications) and throughput-oriented workloads (e.g., games, streaming) according to their computation characteristics (Seo et al., 2015; Zhu et al., 2015a). Our experimental setup covers applications with both throughput-oriented and latency-sensitive bursty computations. We use applications that are commonly used in prior work (Tseng et al., 2014; Zhu et al., 2015b; Pathania et al., 2015). Table 4.5 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* (CamanJS, 2018) is an online image editing application that we execute within *Chrome* web browser. These latency-sensitive applications generate bursts of CPU loads upon user inputs from the GUI. For such applications, the latency of a processing event is the main factor impacting user experience (Endo et al., 1996; Yan et al., 2016) 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* (Aquarium, 2018) and *Rain* (Sheepeuh, 2018) 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-minute 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 (Prakash et al., 2016; Zhu et al., 2015a) as in our work. The FPS metric captures the number of frames that meet the the frame processing deadline (i.e., 18 ms at 60 FPS). Finally, we run the *bodytrack* computer vision application from the PARSEC suite (Bienia et al., 2008) where we monitor the frame-rate (or heartbeats/sec) by instrumenting the application with the Heartbeats framework (Hoffmann et al., 2010). We configure bodytrack to run with the same number of threads with the number of CPU cores in our system (i.e., 8 cores). In order to automate the execution of interactive applications, we use a GUI-based and timing-sensitive record and replay tool (Gomez et al., 2013).

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 seconds, which is a typical duration for interactive sessions (Falaki et al., 2010).

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*[7] of big cores every second if the maximum temperature is lower/higher than 80°C. 80°C is a typical thermal setpoint used in commercial platforms (Prakash et al., 2016; Muller, 2014). 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.

The default DVFS policy in our platform is the *Interactive governor* (Brodowski,

[7]Maximum allowed DVFS state can be altered by modifying the `scaling_max_freq` sysfs entry provided by the cpufreq interface (Brodowski, 2012).

2012), 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 multi-processing (HMP) scheduler (Chung, 2012) 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.

4.3.3 Evaluation of Maestro

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.

Evaluation Methodology: This section states the methodology adopted while conducting the experiments and evaluating the outcomes. We exercise the bursty-

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

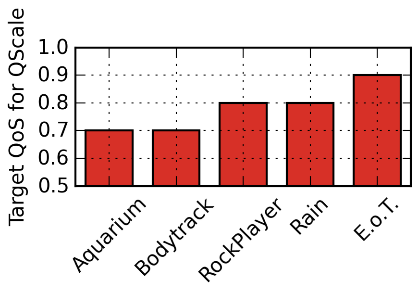


Figure 4·21: Policy selection (left) and QoS targets (right) determined by Maestro for the applications where Maestro detects a continuous throttling-prone computation. Policy selection and QoS setting are based on the methods described in Sections 4.3.1 and 4.3.1, respectively. Maestro assigns lower 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 FPS for *Aquarium*, *RockPlayer*, *Rain* and *Edge of Tomorrow*, respectively.

dominated *Adobe PDF*, *Caman* and *Google Maps* applications for 60-70 seconds, which is the typical length of a user session for many interactive mobile applications (Falaki et al., 2010). We run each throughput-oriented *Aquarium*, *Rain*, *Edge of Tomorrow*, *Rock Player* and *Bodytrack* applications for 10 minutes. 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 minutes before each experiment.

Extending Sustained QoS with Maestro: This section evaluates Maestro's selective QoS tradeoff mechanism as well as quantifying potential improvements in sustained QoS durations achieved on CPU intensive throughput-oriented applications.

The table given in Figure 4·21 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 Figure 4·21. 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 (>3.5W) 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, providing high QoS without sacrificing latency. Such adaptive policy selection capability allows Maestro to make QoS tradeoffs and sustained performance optimization only when necessary.

Figure 4·22 provides an evaluation of the sustained QoS durations achieved by using Maestro and default Android management for the QoS targets described in Figure 4·21. *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

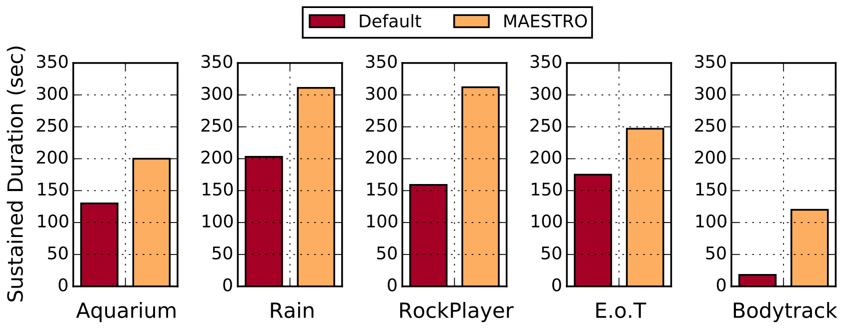


Figure 4·22: Sustained durations achieved by Maestro and default Android management for the QoS targets specified in Figure 4·21.

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. We achieve distinctly longer extensions in sustained QoS for *Rock Player* and *Bodytrack* applications (i.e., 96% and 6.7x, 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.

Figure 4·23 illustrates the thermal profiles of applications with Maestro and default Android management. Maestro achieves lower temperature during the sustained duration. This extra thermal headroom (as high as 15^{o}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 as well as through the thermally-efficient QoS control provided by QScale.

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. We evaluate Maestro's ability to adapt to both inter- and intra-application changes in the computation patterns. For inter-application 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.

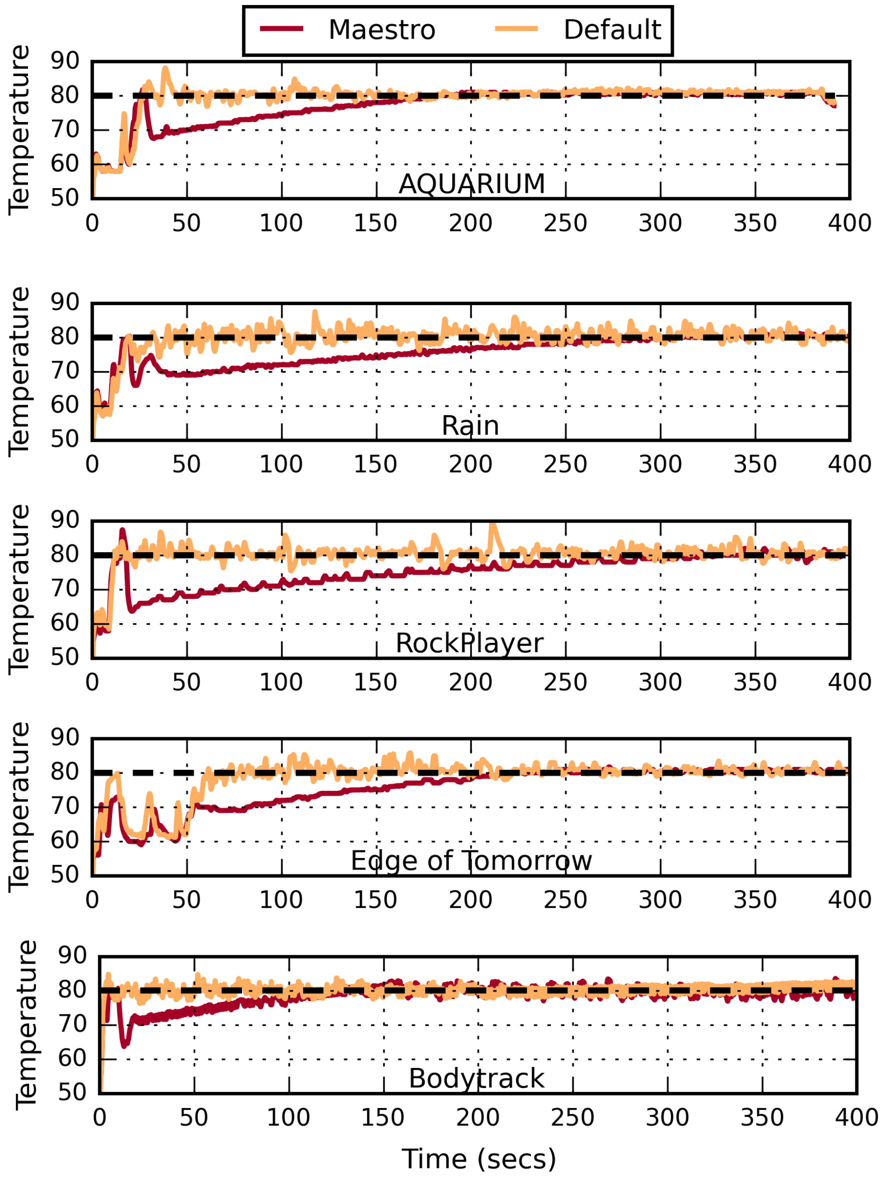


Figure 4·23: Thermal profiles under Maestro and default Android management.

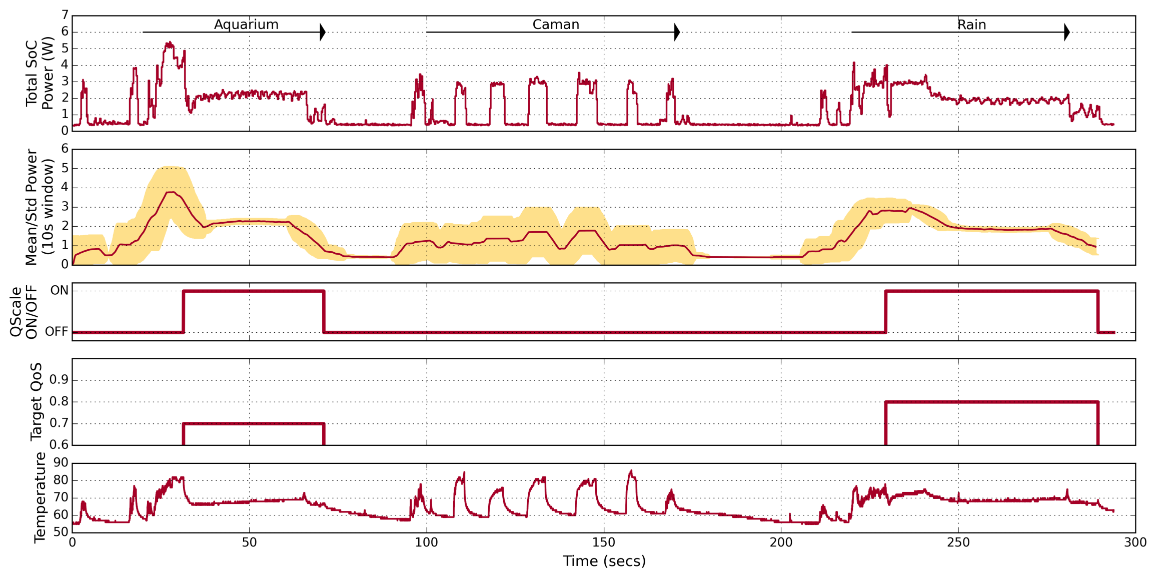


Figure 4·24: Adaptive runtime behavior of Maestro. The user session consists of two throughput-oriented applications with heavy continuous workloads (i.e., *Aquarium* and *Rain*) interleaved by various UI-triggered bursty computations (application launches and image filtering operation in *Caman.js*). Maestro can succesfully 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 10s.

Figure 4·24 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

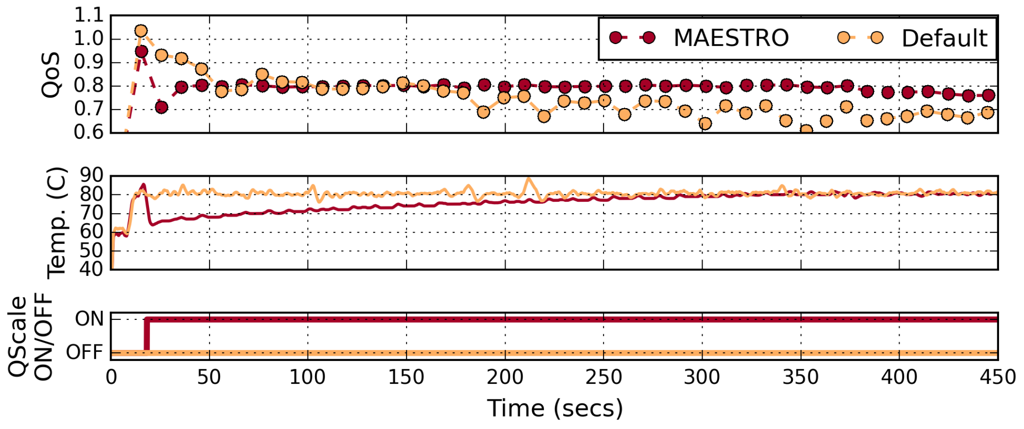
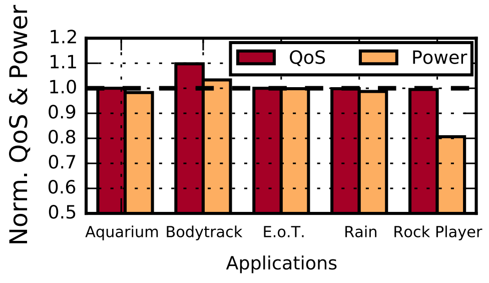


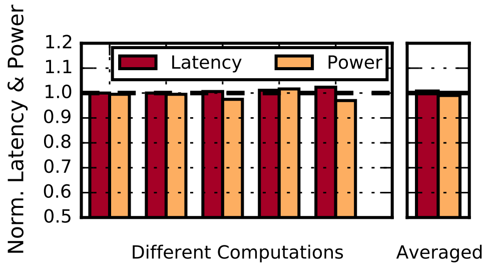
Figure 4·25: Runtime behavior of Maestro for the RockPlayer video application. Maestro detects the heavy continuous computation once the video starts after the initial application launch and the user's menu traversals for video selection. Due to reduced CPU load on big cores with criticality-aware assignment of threads, the QoS degradation is substantially slower after the throttling starts when using Maestro.

after 30 seconds. After 70 seconds, as the user closes the *Aquarium* and launches *Caman* to perform various image editings online, Maestro switches off QScale and hands the 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 Figure 4·24) 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.

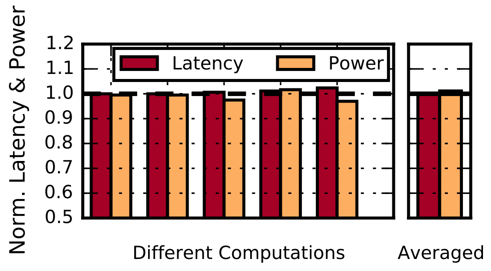
Figure 4·25 provides a detailed look into the *RockPlayer* video player application's runtime profile under Maestro and default Android management. After the application launch and several UI interactions across the application's menu options, the video starts to play and creates a continuously high computation pattern. Maestro detects such pattern and activates QScale after 20 seconds to stabilize the QoS around a 80% target QoS level. Temperature reduces abruptly and throttling does not occur until around 320 seconds, providing almost double the sustained duration



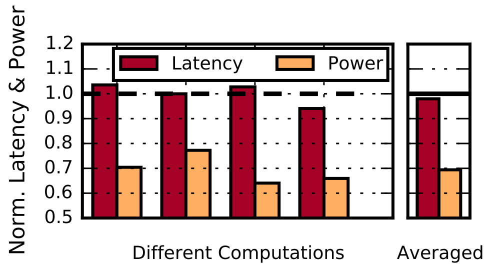
(a)



(b)



(c)



(d)

Figure 4·26: 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*.

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.

Criticality-aware Scheduling vs. HMP: In this section, we verify that the few critical threads identified per application (Section 4.2) 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. Figure 4·26a 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 non-critical 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 4 other worker threads, moving this critical thread onto big cluster as well. We observe that baseline HMP and criticality-aware scheduling 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 Figure 4·26a, Figures 4·26b,4·26c,4·26d show 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. Figure 4·27 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

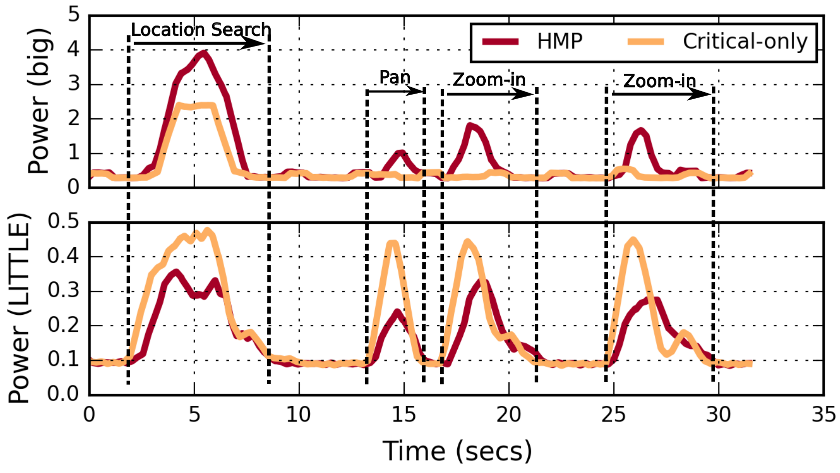


Figure 4·27: CPU power profiles for various computational activities of Google Maps application showing the similar computational latencies at lower power with criticality-aware thread mapping.

ending of the computational activities are captured by detecting the transitions from idle power level as annotated in the figure.

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., 1^{st} block in Figure 4·18). 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

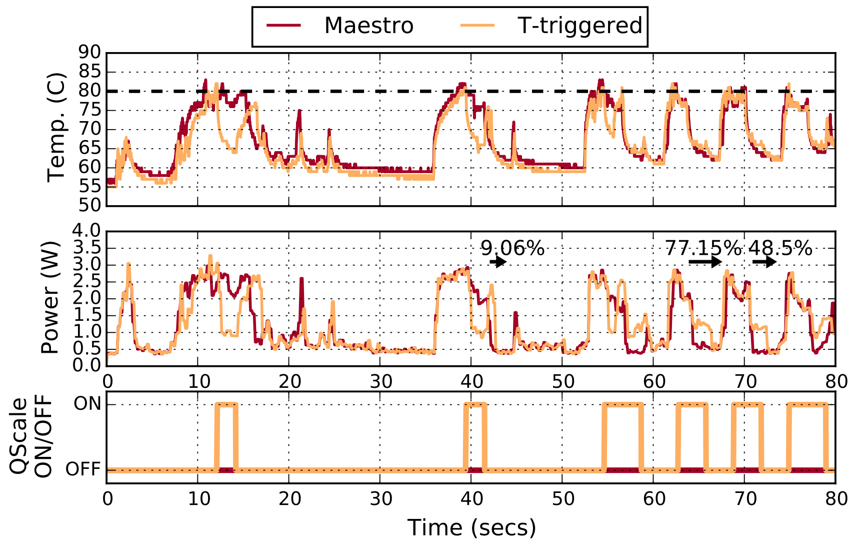


Figure 4·28: Runtime behavior of *Adobe PDF reader* application while operating under Maestro and Temperature-triggered sustained performance control policy. Temperature-triggered policy activates QScale when a critical thermal threshold is hit, and reverts back to default Android management when temperature is below 60°C.

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 Figure 4·28. 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.

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

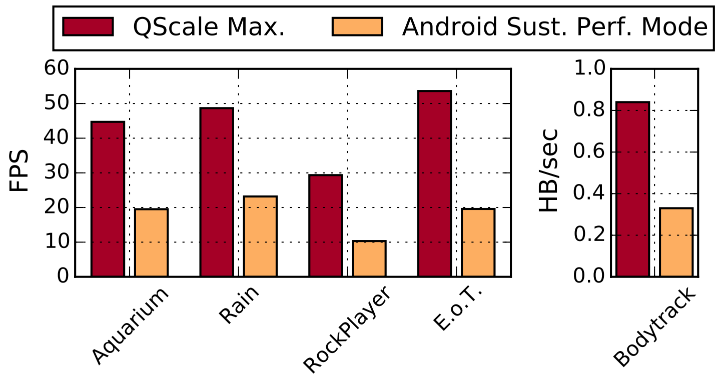
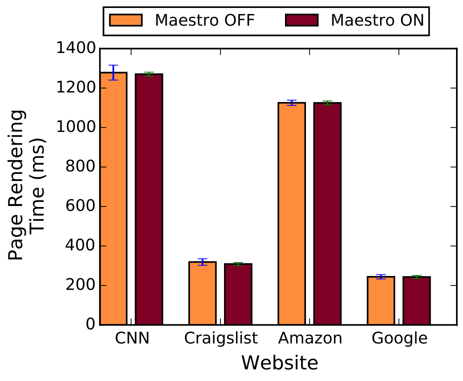
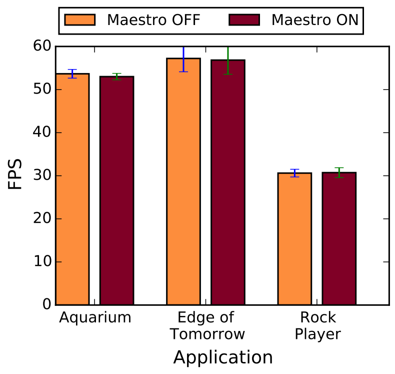


Figure 4·29: A comparison of the maximum attainable QoS for our throughput-oriented applications to QoS levels obtained when operating under Android's sustained performance configuration. Selecting lower power operating modes for CPU and GPU to improve sustained performance with Android sustained performance mode configuration, without any QoS consideration, leads to substantially low QoS for such applications with high computation demand.

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 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 Figure 4·29. 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 FPS, reaching as low as 10 FPS, which would likely deem application unusable from a user experience perspective. Thus, we argue the necessity of QoS consideration for sustained performance management policies.



(a)



(b)

Figure 4·30: Real system measurements to identify whether Maestro policy introduces an overhead that can cause performance degradation on (a) latency-sensitive (b) throughput-oriented applications.

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 (Gutierrez et al., 2011) 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 Figure 4·30, 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`, 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 μsecs (<0.05% of the 1 sec 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 sec sliding window would be 2000*4 bytes (float C type) which occupies only 0.3% of the cache space (2MB) in our platform.

Chapter 5

Software Frameworks for Real-life Studies

Characterizing real-world mobile applications can offer unique insights to optimize current system software. However, studying mobile applications presents major challenges due to their interactive nature that relies heavily on external inputs such as UI events. Unfortunately, due to the challenge of reproducing complex real-life execution of applications, many approaches and prior mobile benchmark suites study mobile applications under limited and unrealistic usage scenarios (e.g., only application launch) (Huang et al., 2014; Pandiyan et al., 2013). In fact, in much of the prior work studying on mobile applications and systems, it often unspecified how such applications are exercised which hinders the reproducibility and comparison of scientific outcomes. Therefore, we argue that the availability of practical approaches that can enable repeatable executions of mobile applications is crucial for systematic analysis and understanding of contemporary mobile workloads. While the UI and network events are common sources of input that dictate the execution of an application, an increasing number of malicious applications alter their execution also depending on the environment they are executed on (e.g., emulated analysis sandbox). Such malware hides their malicious behavior from malware analysis systems for evasion.

This thesis proposes software frameworks to facilitate analysis of real-world mobile applications for efficiency and security. Our first contribution is a record and replay framework, RandR (Section 5.1), to capture and reproduce common sources of non-deterministic inputs such as UI, network events and random numbers. RandR provides record replay capabilities in a practical manner by alleviating the need for any source-level application/OS instrumentation or any privileged modes of operation. Our second contribution tackles the malicious non-determinism in Android applications. We propose a scalable analysis system (Section 5.2) that can automatically identify the heuristics that a malware can leverage to fingerprint underlying malware analysis sandboxes.

5.1 Practical Record/Replay with RandR

While there exist various record/replay tools for desktop and server platforms based on low-level events (e.g., system calls (O'Callahan et al., 2017) or CPU instructions (Patil et al., 2010; Xu et al., 2003)), they are not readily useful for mobile applications. Beyond the need for capturing and replaying multiple non-deterministic factors such UI and network events, accurate record/replay of Android applications also require minimal overhead to preserve timing of events. For instance, the timing between the UI events is crucial to determine the type of the user interaction performed (e.g., (long) click, swipe etc.). While other approaches are specifically geared towards record and replay of mobile applications, several limitations impair their accuracy and practicality. First, most of the previous works (Gomez et al., 2013; Halpern et al., 2015; Hu et al., 2015; Qin et al., 2016) rely on coordinates-based replay which restricts replay capabilities only to one specific device. Moreover, to keep the overhead low, some approaches focus purely on UI events (Gomez et al., 2013; Halpern et al., 2015; Fazzini et al., 2017). Such approaches cannot handle execution variations that arise due to other common factors such as network or random number inputs. Second, much of the prior approaches incur practical drawbacks as they require access to proprietary source code (Espresso, 2019; Fazzini et al., 2017) or require intrusive modifications to underlying OS or virtual machine (Hu et al., 2015). Most real world Android applications are closed-source. Modifying the OS, on the other hand, requires open-source OS and is generally difficult to implement and maintain due to continuously evolving Android software. In addition, due to manufacturer imposed restrictions, it is often impossible to gain the root privileges needed to install a new OS on real world commercial devices.

This thesis proposes the RandR framework which allows for timing sensitive record and replay of multiple sources of inputs while relieving the restrictions that undermine practicality. RandR records the inputs that drive the application execution by *run-time hooking* (as opposed to static OS or VM instrumentation) of a set of target Java and native (C/C++) methods in the Android framework. The runtime hijacking of method calls is a well known technique among security researchers and practitioners (Wong and Lie, 2018; RK700, 2018). However, the key novelty of our work is realizing a system that can record and replay non-deterministic input sources (e.g., GUI and network inputs) in Android applications based on runtime instrumentation in a lightweight and practical manner. RandR hooks into a set of target Java methods to capture user inputs as well as the random numbers that cause variant application behaviour if not kept deterministic across record and replay. In addition, RandR intercepts the network traffic by hooking into standard C libraries.

This section starts with a brief overview of the Android concepts relevant to understanding RandR (Section 5.1.1). In Sections 5.1.2 and 5.1.3, we present an overview and implementation of our RandR framework, respectively. Finally, Section 5.1.4 presents an evaluation of RandR with real-world Android applications.

5.1.1 Background

This section briefly reviews the Android concepts that are necessary to understand our record and replay system. Specifically, we discuss how applications are distributed and executed on the Android platform. Next, we describe Android's input handling mechanism. Finally, we describe the principles of hooking into Java APIs in `ART`.

Android System: Android applications are mainly written in Java on top of the APIs provided by the Android framework, compiled into Dalvik bytecode and executed by the Android Runtime (`ART`). These applications are distributed in the form of `APK` files that contain an application's bytecode (`.dex`), resources, assets as well as a manifest file. The manifest file declares a set of permissions that grants the application with access to additional functionality (e.g., network, storage). The Android platform is built upon a modified version of the Linux kernel and inherits the same user-based permissions and process models to provide each application with an isolated execution environment. Each application runs in its own process and with its own instance of the `ART`.

Android applications can also implement a part of their functionality in native code (e.g., in `C/C++`). Android framework also consists of a set of native libraries that provide low-level system functionalities (e.g., `libssl` for SSL support, `libgles` for 2D and 3D graphics rendering).

Android UI System and Input Handling: User interactions over the touchscreen is an important source of inputs for Android applications. In Android, UI events are described via MotionEvent and KeyEvent classes, collectively described as InputEvent. MotionEvents specify the UI input in terms of an action code (e.g., `ACTION_UP`, `ACTION_DOWN`) and screen coordinates. Complex gestures (e.g., fling, pinch) are described as a sequence of MotionEvents. KeyEvents contain information about a key that has been pressed. To facilitate design of user interfaces, the Android framework provides a rich variety of essential UI elements (*widgets*) such as Buttons, TextViews, ImageView or ScrollView. Moreover, a developer is able to integrate custom widgets by overriding View or ViewGroup classes. Views are user interface elements that represent interactive objects on the screen, while ViewGroup is responsible for organizing Views and other ViewGroups into a layout tree. An application can chose to define its layout tree statically, dynamically, or both.

Android framework also describes window and root view for organizing various interactive screen of an app. A window in Android is rectangular area on the screen where an Activity can draw its UI interface. All UI components in a window form one view hierarchy with the root view defined in ViewRootImpl class. This class describes the behavior of the window and has two primary functions relevant to our goals. First, each instance of ViewRootImpl class registers an InputEventReceiver, a low level mechanism that is used to deliver all InputEvents to an application window. Second, this class traverses the view hierarchy to determine which View will receive user input.

Dynamically Instrumenting Java APIs in `ART`: Each `Java` class is internally represented by a `Class` object in memory in `ART`. A virtual method table (i.e., vtable) within a `Class` is used to resolve virtual methods and implement polymorphism. In `ART`, the `ArtMethod` structure corresponds to internal C++ representation of both static and virtual Java methods. Thus, vtable of a `Class` basically points to `ArtMethod` objects in memory. Each `ArtMethod` object contains a set of *entry points* which are essentially pointers to code that performs necessary set-up/clean-up and executes the target method's code. ART uses different entry points depending on the method type (e.g., Java or native). Hooking[8] in `ART` can be accomplished either by modifying an entry in the vtable to point to a different `ArtMethod` object (Costamagna and Zheng, 2016) or by modifying entry point within an `ArtMethod` to point to a different location (Wong and Lie, 2018). The former approach allows to hook into virtual methods. RandR adopts the entry point hooking approach which works with both static and virtual methods.

[8]We use the term hooking and dynamic instrumentation interchangeably.

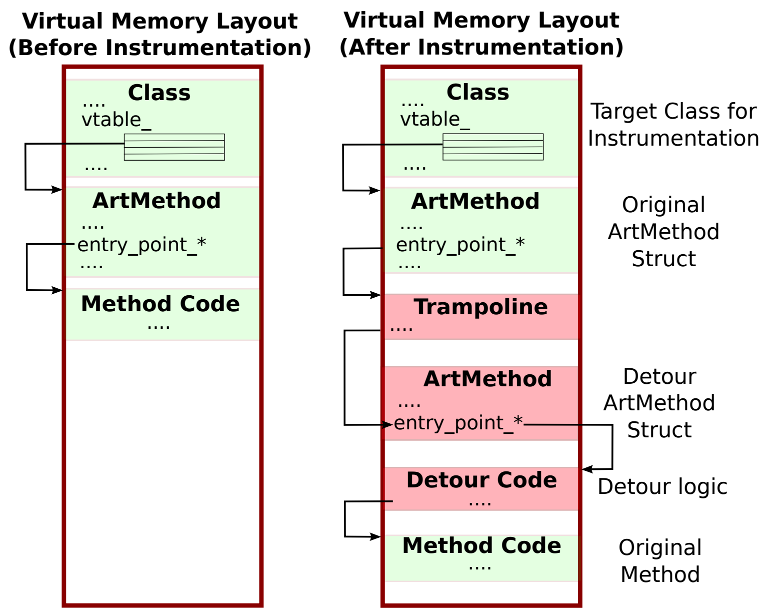


Figure 5·1: Dynamic instrumentation in memory. Green regions represent the original class and method structures in memory while red regions highlights the dynamically injected components.

The reflection API support in JNI can be used to obtain a handle to `ArtMethod` object for a target method of interest. Once the address of the target `ArtMethod` struct is known, the entry point can be accessed by a fixed offset and modified to point to a different address. This allows to execute a different code upon invocation of the target method. We refer to this newly injected code as the *detour* method. Once the control is diverted to the detour function from the target method, the method arguments can be recorded or modified. The original entry point of the target method can also be invoked from the detour function to implement the original method call behavior. Once the original target method is executed, the return values can also be recorded or modified as necessary. Figure 5·1 summarizes this process by illustrating the state of the virtual memory layout of an application before and after the dynamic instrumentation.

5.1.2 RandR Overview

The goal of RandR is to provide practical cross-device record and replay capabilities for Android applications. In case of RandR, practicality is achieved by alleviating the limitations that hinder deployability such as root permissions or modifications to OS/application source. RandR realizes these capabilities through a combination of static application instrumentation and dynamic framework instrumentation that allows us to capture and reproduce the inputs to a set of target method calls (i.e., both `Java` and `C` APIs) in the Android framework. This section provides an overview of these static and dynamic instrumentation components. It also describes working principles of RandR (Figure 5·2) and our approach to cross-device replay.

The purpose of our static application instrumentation stage is to enable dynamic framework instrumentation from within the app. RandR decodes the application resources and `dex` files, rewrites application's `bytecode` so as to load and execute our dynamic instrumentation component at runtime, and generates a new `APK` file. Performing the framework instrumentation at runtime and from within the application provides RandR with unique advantages. First, the target method structures can be identified and patched in memory at runtime without requiring any static OS modifications or access to any device software. Second, since target method structures are

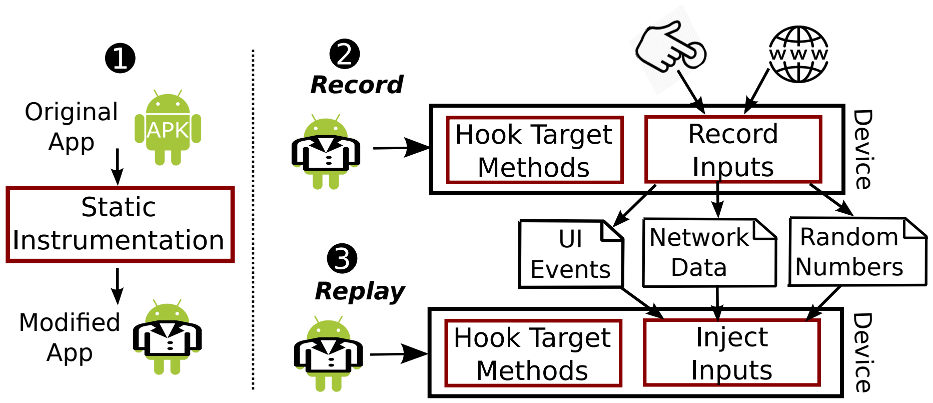


Figure 5·2: Macro view of our RandR record/replay approach.

introspected by the application in its own process address space, RandR does not require any privileged mode operation or root permissions.

Dynamic instrumentation is the key component of RandR that *hooks*[9] into a set of target method structures in memory to capture (i.e, for record) or modify (i.e., for replay) inputs and return values. Our aim is to capture and reproduce the inputs to the application that originate from external non-deterministic sources such as random numbers, UI interactions and network. During the recording phase, we direct the control flow to custom *detour* methods that record inputs into a trace file. RandR also records timestamps to ensure that the timing between the method invocations are preserved during replay. Replay logic differs depending on whether the target method invocation originated from the Android runtime (i.e., *upcall*) or the application (i.e., *downcall*) while recording. For the upcalls, RandR invokes the original method with saved arguments during replay. Since the downcalls that originate from the application are diverted to our detour methods after instrumentation, for such calls, RandR simply modifies the original method arguments with those from the trace file.

In order to provide cross-platform replayability of recorded UI events, we present a widget-sensitive record and replay approach. RandR dynamically instruments strategic locations in the Android framework to associate InputEvents with their target UI widget (e.g., a button) and assigns each widget a stable identifier. Thus, during replay, RandR is able to locate a target widget and send UI events to accurate positions in the device screen accordingly.

5.1.3 RandR Implementation

This section describes our implementation of the RandR framework. We first describe our static application instrumentation mechanism. Next, we detail how we

[9]We use the term hooking and dynamic instrumentation interchangeably in this thesis.

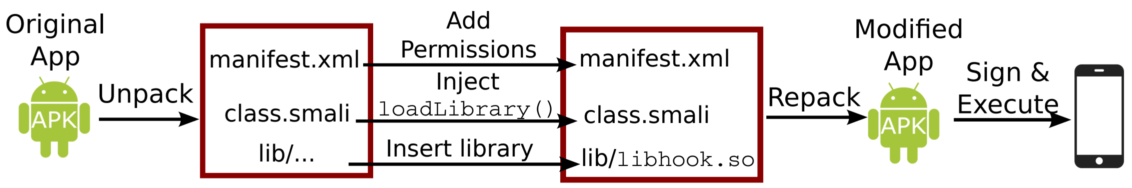


Figure 5·3: Static instrumentation steps in RandR.

leverage our hooking mechanism to enable record and replay of UI, network and random number inputs for Android applications.

Static Application Instrumentation

RandR's static instrumentation component (Figure 5·3) modifies the application resources and bytecode so as to inject hooks into target method structures within applications virtual address space. We use `apktool`[10] to disassemble the bytecode from application's `dex` files and obtain the manifest file (`AndroidManifest.xml`). The disassembled bytecode is in readable and editable `smali`[11] representation. To ensure our hooks are in-place before the application starts to execute, RandR injects a small stub code into the `Application` subclass of the app. We choose this approach as the `Application` subclass in Android is instantiated before any other class. The small stub code simply invokes `System.loadLibrary()` to load our native hooking library, which we insert into the `APK`. RandR also modifies the manifest file to add missing storage access permissions to allow the application to read/write trace files.

Dynamic Runtime Instrumentation

This section details how RandR realizes record and replay capabilities for Android applications by hooking into a set of target method structures in memory. We also describe the implementation of our hooking library.

[10]`https://ibotpeaches.github.io/Apktool/`
[11]`https://github.com/JesusFreke/smali`

Hooking into Java and Native APIs: We build our Java API instrumentation setup on top of an open-source hooking framework for `ART` (i.e., `YAHFA` (RK700, 2018)). As per the entry point hooking process described in Section 5.1.1, `YAHFA` framework creates a back-up of the original `ArtMethod` object and replaces the target of the entry point with a trampoline that simply jumps to the entry point of the `ArtMethod` object corresponding to our detour method. The original method can be executed simply by invoking the backed-up original `ArtMethod` from our detour functions.

Our implementation of this hooking infrastructure is fully contained in a shared native library and leverages the reflection API support in JNI to obtain references to method instances at runtime (e.g., using `GetStaticMethodId()` routine). Our framework uses dynamic class loading capabilities provided by the Android framework (i.e., via `DexClassLoader`) to dynamically load detour methods (written in Java) as well as other replay components into the virtual memory space of the target application.

Our implementation of native library hooks uses the `AndHook` library(Lody, 2018) which provides a hooking interface for various Application Binary Interfaces (`ABIs`) including armeabi-v7a, arm64-v8a, x86 and x86_64. Hooking a native function involves locating the image of the shared library in the process address space, parsing the `ELF` structure to identify the address of the target functions and inserting necessary jumps to implement the detour mechanism.

Recording and Replaying UI Inputs: RandR's approach to UI replay is based on the observation that a user mostly interacts with an application via UI widgets, essential UI elements provided by the Android Framework (e.g. `Button`, `SearchBar`, `ImageView`, etc). Therefore, RandR targets widget-sensitive UI replay, and records the UI events (e.g., `MotionEvent`, `KeyEvent`) with respect to an application's UI widgets. To uniquely identify widgets presented in an app, RandR assigns widgets stable identifiers that are consistent across multiple application runs and devices.

RandR derives these identifiers from multiple internal properties of a widget, including the widget's position in the UI layout, text and other internal fields. During the replay, RandR identifies the widgets on the screen, updates the recorded events according to the new location of a widget, and injects updated events into the app.

Recording and Replaying Network I/O: Our RandR framework intercepts the network I/O by intercepting the system call wrapper methods in Android's standard C library (`libc`) implementation called `Bionic`. By performing the record and replay at the system call interface, RandR is agnostic to different network protocol implementations (e.g., OkHttp, Volley).

Unlike protocols such as `HTTP` and FTP, recording and replaying the results of system calls is not sufficient to reproduce an `HTTPS` traffic, which is widely popular among applications. The challenge rises due to inherent non-deterministic nature of the cryptographic protocols (e.g., `TLS`) used in `HTTPS`. Our key insight for tackling this problem is that the root cause of non-determinism is due to the random numbers used for generating the session keys during the `TLS` handshake process. RandR inherently captures the server-side random inputs through system calls (e.g., `read()`). For the client-side random numbers, RandR hooks into cryptographic libraries.

Instrumented Lib	**Instrumentation Points**
libc.so	*socket(), connect(), read(), write(), close(), poll(), sendto(), recvfrom(), shutdown()*
libcrypto.so	*RAND_bytes()*

Table 5.1: Target native instrumentation points in RandR.

Table 5.1 provides the specific set of methods RandR instruments for network record and replay. During the record phase, we invoke the original function from our detour method, group the arguments and return values with respect to the file descriptor and record to a trace file. During the replay phase, once a `connect()` call is made on a socket, we identify the corresponding trace file to read from based on the target `addr` argument. Following operations on this socket (e.g., `read()`, `recv()`, `recvfrom()`) read the data into argument buffer from the corresponding trace file. `write()`, `send()` and `sendto()` operations are discarded during replay as the application is not communicating with a server. To enable recording and replaying the `HTTPS` traffic, RandR fills the argument buffer in the native `RAND_bytes` crypto API method with a fixed set of data.

Recording and Replaying Random Numbers Random numbers present another significant source of non-determinism in Android applications which, if not captured and replayed, can lead to different application behaviors (Continella et al., 2017). We note that such random input dependent variations could even be intentional from the developer's perspective (e.g., 2048 gaming app).

RandR hooks into the pseudo-random number generator in the Java API. Specifically, we record the return values from the `next()` method in the `java.util.Random` class. We choose this specific instrumentation point since it is the common subroutine among other public random number APIs (e.g., `Random.nextInt()`, `Random.nextBytes()`, `Math.random()`). During replay, return values of the `next()` method is overridden by the next number in the sequence of recorded values.

5.1.4 Evaluation

We evaluate RandR's replay accuracy via two metrics: (1) Jaccard similarity between the set of executed methods of an application during record and replay; (2) user-visible state similarity as used in prior work (Hu et al., 2015; Halpern et al., 2015; Gomez et al., 2013). We use a common `Java` code coverage tool (i.e., `EMMA` (EMMA, 2006)) to obtain the set of executed methods. Since `EMMA` requires the application's source code for instrumentation, we crawl a random set of real-world Android applications from the F-droid dataset, and select 10 applications that `EMMA` and RandR can instrument without any errors.

	Jaccard Similarity		Visual Similarity			Jaccard Similarity		Visual Similarity	
App	RandR	Reran	RandR	Reran	**App**	RandR	Reran	RandR	Reran
Knights of Alentejo	100%	100%	Yes	Yes	Summation	100%	94%	Yes	No
Solar Compass	97%	97%	Yes	No	Accordion	100%	97%	Yes	Yes
Verbiste Android	97%	80%	Yes	Yes	Word Power	100%	98%	Yes	No
ToneDef	100%	100%	Yes	No	MedicLog	100%	41%	Yes	No
Mileage	100%	47%	Yes	No	Dicer	100%	94%	Yes	No

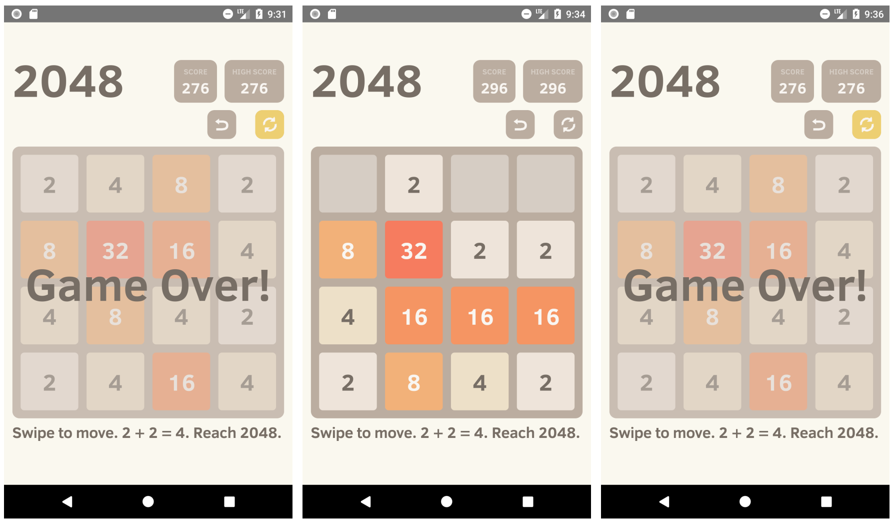
Table 5.2: Cross-device evaluation of RandR and Reran(Gomez et al., 2013)

We evaluate RandR's cross device replay capability on two Android SDK emulators (Nexus 5X and Pixel 2 XL) that have different screen sizes. Overall, when exercising applications during record, we achieve an average 49.45% method coverage. As shown in Table 5.2, RandR was able to successfully replay all applications across both devices, unlike the coordinates-based Reran tool.

To assess RandR's ability to record and replay closed-source applications, we pick the most popular applications with network dependent functionalities (i.e., using both `HTTP` and `HTTPs`) from 4 Play Store categories: OfficeSuite, Kakao Bus, Mirror Camera, Hot Pepper Gourmet. For these applications, we verified RandR's replay success by comparing the recorded and reproduced user-visible screen states.

We show the importance of handling inputs other than UI (e.g., random numbers) with a specific study comparing RandR to Reran (Gomez et al., 2013) for a session of the 2048 game (Figure 5·4). During record, the game reaches the "Game Over" state and renders a new text on the screen. Replay with Reran diverges to a different state due to randomized location of numbers, while RandR reproduces the same sequence of random numbers and reaches the correct final state.

Impact of Accurate Replay on Performance Measurements: RandR allows for real-life experimentations (e.g., power or performance studies) on mobile systems with off-the-shelf mobile applications and can substantially improve the quality of experimental measurements. Figure 5·5 illustrates the significance of replaying net-


2048
SCORE 276
HIGH SCORE 276
Game Over!
Swipe to move. 2 + 2 = 4. Reach 2048.
2048
SCORE 296
HIGH SCORE 296
Swipe to move. 2 + 2 = 4. Reach 2048.
2048
SCORE 276
HIGH SCORE 276
Game Over!
Swipe to move. 2 + 2 = 4. Reach 2048.

(a) Record **(b)** Reran **(c)** RandR

Figure 5·4: Comparison of Reran (Gomez et al., 2013) to RandR for 2048 application

work traffic (e.g., using RandR), as opposed to UI-only replay (Gomez et al., 2013; Halpern et al., 2015; Fazzini et al., 2017; Qin et al., 2016), for achieving consistent and meaningful performance measurements. We measured the latency distribution over 20 executions of `UC Browser` application from the Play Store while browsing a web page whose content changes over time (i.e., `thefakenewsgenerator.com`). Since RandR replays the same recorded content and is not effected by the network speed fluctuations or the web content changes on the server, RandR can significantly reduce measurement variations. Such reproducibility is key to correct analysis of performance and power bottlenecks in mobile systems research.

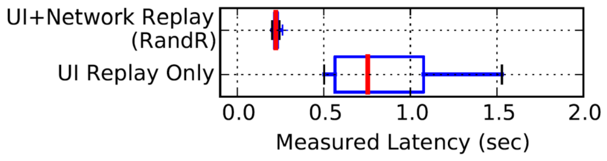


Figure 5·5: Performance variance with and without network replay

5.2 Detecting Discrepancies in System Emulators

Android is an attractive target for cyber criminals due to billions of users worldwide. To protect users, malware analysis systems which inspect applications in an emulated sandbox are commonly used in academia and industry (Yan and Yin, 2012; Tam et al., 2015; Oberheide and Miller, 2012). The effectiveness of these dynamic malware analysis systems, however, is largely at risk due to an emerging class of evasive malware. Such malware looks for discrepancies that exist between emulated and real systems before triggering any malicious attempt. By ceasing malicious activities on an emulated enviroment, the malware can thwart existing emulator-based malware analyzers. The situation is alarming as studies show a rising number of malware instances that employ evasion tactics (Lindorfer et al., 2011) (e.g., Branco et al. find evasion methods in more than 80% of 4M malware samples (Branco et al., 2012)). For Android, several recent classes of evasive malware (e.g., *Xavier* (Xu, 2017), *Grabos* (Davis, 2017)) have already been identified in the Play Store. A crucial step for defending against such malware is to systematically extract the discrepancies between emulated and real systems. Once discovered, such discrepancies can be eliminated (Liu et al., 2017) or can be used to inspect applications for presence of evasion tactics leveraging these artifacts (Branco et al., 2012).

Many of the approaches to date (Thuxnder, 2015; Petsas et al., 2014; Vidas and Christin, 2014) discover discrepancies of emulation-based sandboxes in an ad hoc fashion by engineering malware samples or specific emulator components (e.g., scheduling). Such manual approaches cannot provide large-scale discovery of unknown discrepancies, which is needed to stay ahead of adversaries. Recent work (Jing et al., 2014) automatically identifies file system and API discrepancies used by several Android malware (e.g., (Xu, 2017; Davis, 2017)). Evasion tactics that rely on such artifacts can be rendered ineffective by using modified system images and ensuring the API return values match those in real devices (Bordoni et al., 2017). Besides API/file checks, a malware can also leverage differences in the semantics of CPU instructions to fingerprint emulation (Branco et al., 2012) (e.g., by embedding checks in the native code (Petsas et al., 2014)). As opposed to ad hoc approaches or API/file heuristics, our work focuses on systematically discovering instruction-level discrepancies that are intrinsically harder to enumerate and fix for complex CPUs.

This thesis proposes the Proteus system which identifies the discrepancies between emulated and real ARM CPUs that power the vast majority of current mobile devices. Unlike prior discoveries of instruction-level discrepancies (Paleari et al., 2009; Martignoni et al., 2009), Proteus allows horizontal scaling by alleviating the real hardware requirement and reveals the *root causes* behind the discrepancies. As a result, we have discovered and fixed several root causes of discrepancies in a state-of-the-art emulator (i.e., QEMU) used by Android malware analyzers. This section starts by providing some critical background for Proteus. Sections 5.2.2 and 5.2.3 presents an overview and implementation of our Proteus system. Section 5.2.4 evaluates Proteus' ability to discover discrepancies between real and emulated ARM CPUs.

5.2.1 Background

This section provides a brief overview of the ARM architecture and clarifies the terminology that we use throughout the rest of this paper. We also describe the attack model we are assuming in this work.

ARMv7-A Architecture

This paper focuses on ARMv7-A instruction set architecture (ISA), the vastly popular variant of ARMv7 that targets high-performance CPUs which support OS platforms such as Linux and Android (e.g., smartphones, IoT devices). The ARM architecture implements a Reduced Instruction Set Computer (RISC) organization where memory accesses are handled explicitly via load/store instructions. Each ARM instruction is of fixed 32-bit length. ARMv7-A features 16 32-bit registers (i.e., 13 general purpose registers (`R0-R12`), stack pointer (`SP`), link register (`LR`), program counter (`PC`)) accessible in user-mode (`usr`) programs. The CPU supports 6 operating modes (`usr`,`hyp`,`abt`,`svc`,`fiq`,`irq`) and 3 privilege levels `PL0`, `PL1` and `PL2` (i.e., lower numbers correspond to lower privilege levels). The *Current Program Status Register* (`CPSR`) stores the CPU mode, execution state bits (e.g., endianness, ARM/Thumb instruction set) and status flags.

Undefined Instructions: The ARMv7 specification explicitly defines the set of encodings that do not correspond to a valid instruction as architecturally `Undefined`. For example, Figure 5·6 shows the encoding diagram for multiplication instructions in ARMv7. The architecture specification (ARM, 2018b) states that the instructions are `Undefined` when the `op` field equals 5 or 7 in this encoding.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0	op		0 0 0 0	

Figure 5·6: Encoding diagram for multiplication instructions in ARMv7 ISA (ARM, 2018b).

An `Undefined` instruction causes the CPU to switch to the undefined (`und`) mode and generates an undefined instruction exception. An undefined instruction exception is also generated when an instruction tries to access a co-processor that is not implemented or for which access is restricted to higher privilege levels (ARM, 2018b).

Unpredictable Instruction Behavior: The ARM architecture contains a large set of instruction encodings for which the resulting instruction behavior is unspecified and cannot be relied upon (i.e., `Unpredictable`). ARM instructions can exhibit `Unpredictable` behavior depending on specific cases of operand registers, current CPU mode or system control register values (ARM, 2018b). For example, many instructions in the ARM architecture are `Unpredictable` if the `PC` is used as a register operand. In addition, some instruction encoding bits are specified as *"should be"* and denoted as "(0)" and "(1)" in ARM's official encoding diagrams. While different encodings for *"should be"* bits do not correspond to different instructions, the resulting behavior is `Unpredictable` if a given encoding fails to match against the specified *"should be"* bit pattern.

The effect of an `Unpredictable` instruction is at the sole discretion of the CPU manufacturer and can behave as a `NOP` or `Undefined` instruction, or can change the architectural state of CPU. Consider the "`LDMDA pc!,{r0,r1,r5,r6, r8,sp,lr}`" `Unpredictable` instruction (encoded as `0xE83F6163`), which loads the given set of registers from consecutive memory addresses starting at `PC` and writes the final target address back to `PC`. This instruction causes undefined instruction exception on a real CPU while it modifies the `PC` and causes an infinite loop on QEMU. Note that both behaviors comply with the ARM specification.

Threat Model

The aim of the malware author is to evade detection by the analysis tools and distribute a malicious application to real users. The malware possesses a set of detection heuristics to distinguish emulators from real devices. Malware achieves evasion by ceasing any malicious behavior on an emulated analysis environment, which could otherwise be flagged by the analysis tool. Once the malware escapes detection and reaches real users, it can execute the harmful content within the application or dynamically load the malicious payload at runtime (Poeplau et al., 2014).

Our work focuses on discrepancies that are observable by user-level programs. Thus, we assume applications running in `usr` mode at the lowest `PL0` privilege level. Since our technique detects emulators by natively executing CPU instructions and monitoring their effects, we assume an Android application that contains a native code. This is a common case for many applications (e.g., games, physics simulations)

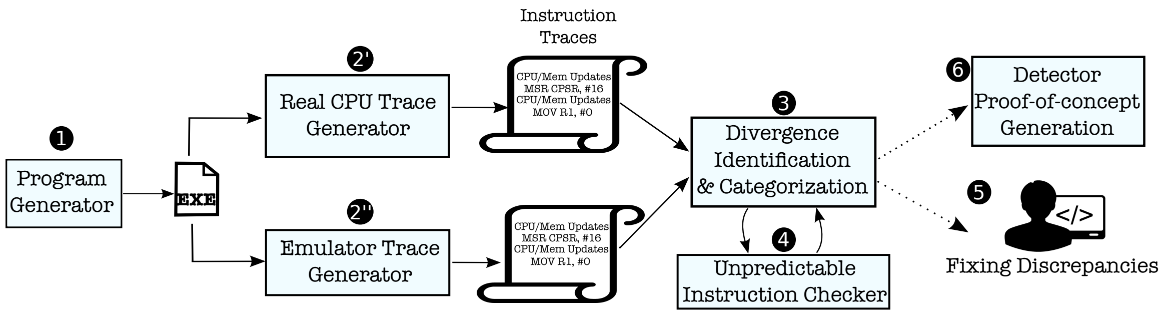

Instruction Traces
2'
Real CPU Trace Generator
CPU/Mem Updates MSR CPSR, #16 CPU/Mem Updates MOV R1, #0
1
Program Generator
EXE
2''
Emulator Trace Generator
CPU/Mem Updates MSR CPSR, #16 CPU/Mem Updates MOV R1, #0
3
Divergence Identification & Categorization
4
Unpredictable Instruction Checker
6
Detector Proof-of-concept Generation
5
</>
Fixing Discrepancies

Figure 5·7: Overview of PROTEUS.

that use native code for the performance-critical sections and for the convenience of reusing existing C/C++ libraries (Poeplau et al., 2014).

We assume that applications are subject to dynamic analysis in a QEMU-based emulation environment. Indeed, state-of-the-art dynamic analysis frameworks that are commonly used in academia (Yan and Yin, 2012; Tam et al., 2015) and industry (Oberheide and Miller, 2012) use QEMU as the emulation engine. In addition, the Android emulator that is distributed with the Android SDK is also based on QEMU.

5.2.2 Proteus System Architecture

The aim of the proposed PROTEUS system (Figure 5·7) is to find the differences in semantics of instructions executed on a real and an emulated ARM CPU. PROTEUS consists of a trace collection part and an analysis component to automatically identify and classify divergences. This section provides an overview of the core components of PROTEUS and describes its high-level operation.

Central to our system is collection of detailed instruction-level traces that capture the execution behavior of programs on both emulated and real CPUs. The traces capture all updates to user-visible registers as well as the operands in memory transactions from load/store instructions. If a program terminates by a CPU exception, the respective signal number is also recorded.

The *"Program Generator"* component (❶) generates the test programs which are used for collecting instruction-level traces and discovering discrepancies. Note that ARM CPU emulation in QEMU is inadvertently tested using millions of applications by Android developers. Thus, programs generated for divergence identification should also exercise platforms for uncommon cases beyond the set of instructions emitted by compilers and found in legitimate Android applications.

For each generated test program, we collect its instruction-level traces by executing the same binary on two different platforms (❷) which provide the traces corresponding to execution on an emulator and a real CPU.

The *"Divergence Identification & Categorization"* component (❸) compares emulator and real CPU traces of a program to identify the initial point of divergence. A divergence can be due to a mismatch in register values, memory operands or exception behavior. Divergent cases that stem from the same mismatch are grouped together automatically to facilitate manual inspection of discovered discrepancies. Our hypothesis behind the grouping is that there exist a small number of root causes that cause the same divergent behavior (e.g., exception mismatch) on potentially a large set of test cases. For instance, we can group together the divergent instructions that generate an illegal instruction exception in a real CPU but execute as a valid instruction in emulator. We also check if the divergent instruction is `Unpredictable` (❹). Since `Unpredictable` instructions can exhibit different behavior across any two platforms, we do not treat divergences that stem from these instructions as a reliable detection method.

Overall, PROTEUS provides us with the instruction encoding that caused the divergent behavior, register values before that instruction, divergence group as well as the difference between the traces of emulated and real CPU (e.g., signal number, CPU mode, etc.) which occurs after executing the divergent instruction. We can optionally identify why QEMU fails to faithfully provide the correct behavior as implemented by the real CPU and fix the source of mismatch (❺). Proteus can also generate a proof-of-concept emulation detector (❻), which reconstructs the divergent behavior by setting respective register values, executing the divergent instruction and checking for the resulting mismatch that Proteus identifies during the *"Divergence Identification & Categorization"* stage.

5.2.3 Proteus Implementation

In this section, we describe our implementation of the proposed Proteus system for detecting instruction-level differences between emulated and real ARM CPUs. We first describe our framework for acquiring instruction-level traces followed by details on how we use this framework to collect a large number of sample traces and automatically identify discrepancies.

Instruction-level Tracing on ARM-based Platforms

Collected Trace Information: For our purposes, a trace consists of all general-purpose registers that are visible to user-level programs, which provide a snapshot of the architectural state. Specifically, we record the `R0-R12`, `SP`, `PC`, `LR` and `CPSR` registers (see Section 5.2.1). Finally, we record operands of all memory operations. Various ARM instructions can load/store multiple registers sequentially from a base address. We record all the data within the memory transaction as well as the base address. This trace information gives us a detailed program-visible behavior of CPU instructions. Thus, any discrepancy within the trace is visible to a malware and can be potentially leveraged for evasion purposes.

Emulator Traces through QEMU Instrumentation: QEMU dynamically translates the guest instructions (e.g., ARM) for execution on the host machine (e.g., x86). Translation consists of several steps. First, guest instructions within a basic block are disassembled and converted into a platform-agnostic intermediate representation called TCG (*Tiny Code Generator*). Next, generated TCG code blocks (i.e., *translation block*) are compiled into host ISA for execution.

To implement tracing capability in QEMU, we inject extra TCG operations into each translation block during the translation phase. These extra TCG operations dump the trace information during the execution phase. We use the helper functionality within QEMU to generate the extra TCG code. The main use of the helper functionality in QEMU is to allow developers to extend the capabilities of TCG operations for implementing complex instructions. We inject the extra TCG operations for every disassembled instruction to achieve per-instruction tracing granularity. Specifically, we modify the disassembly routines of ARM instructions to inject TCG operations that record registers. We also modify the load/store routines to record address and data values for memory transactions.

We use QEMU 2.7.0 from Android repositories[12], which forms the base of the SDK emulator used in modern Android malware analyzers (Yan and Yin, 2012; Oberheide and Miller, 2012; Tam et al., 2015). QEMU 2.7.0 is the most recent version adopted in current SDK emulators. To ease instrumentation and facilitate the data collection, we use QEMU in user-mode configuration as opposed to full-system emulation. We use full-system SDK emulators during our evaluation of discovered discrepancies.

Accurate Real CPU Traces using ARM Fast Models: Gathering detailed instruction-level traces from real CPUs is challenging and, due to practical limitations on the number of devices that can be used, does not scale well. In this work, we propose to use accurate functional models of ARM CPUs (i.e., *Fast Models* (ARM, 2018a)) to obtain traces corresponding to execution on real CPUs. Fast Models are official software models developed and maintained by ARM and provide complete accuracy of software-visible semantics of instructions.

ARM Fast Models provide a set of trace sources which generate a stream of trace

[12]https://android.googlesource.com/platform/external/qemu-android/+/qemu-2.7.0

events when running the simulation. Once a target set of trace sources are specified, Fast Models emit trace events whenever a change occurs on a trace source. These trace events are provided over a standardized interface called *Model Trace Interface (MTI)*. We use an existing plugin called *GenericTrace* to record trace events over the MTI interface.

Our work is based on a Cortex-A15 fast model which implements the ARMv7 ISA. We specify "*inst*", "*cpsr*", "*core_loads*", "*core_stores*" and "*core_regs*" trace sources, which capture changes in register values as well as data/address operand values in memory transactions.

Figure 5·8 shows an example trace snippet we collect from the Fast Model for an `LDR` instruction which loads a word from memory into `SP`. The model emits two trace sources, "*core_loads*" and "*core_regs*", capturing the address/data operands of load operation as well as the update on `SP` (e.g., old and new value). The "*inst*" trace event also specifies the executed instruction (e.g., `PC`, byte code, mnemonic).

```
1 INST|PC=0x0000807c|OPCODE=0xe59fd01c|SIZE=0x04|MODE=usr|ISET=ARM|PADDR=0x000000000000807c|NSDESC=0x00|
  PADDR2=0x000000000000807c|NSDESC2=0x00|NS=0x00|ITSTATE=0x00|INST_COUNT=0x0000000000000003|CORE_NUM=0x00|
  DISASS="LDR      sp,{pc}+0x24 ; 0x80a0"

2 CORE_LOADS|VADDR=0x000080a0|RESPONSE=OK|LOCK=Normal|TRANS=N|ACQREL=None|SIZE=0x04|ELEMENT_SIZE=0x04|
  PADDR=0x00000000000080a0|NSDESC=0x00|PADDR2=0x00000000000080a0|NSDESC2=0x00|DATA=0x0xbefffb30:4

3 CORE_REGS|ID=0x0d|PHYS_ID=SP_usr|VALUE=0xbefffb30|OLD_VALUE=0x00000000|MODE=usr|CORE_NUM=0x00
```

Figure 5·8: A sample trace from Cortex-A15 Fast Model for an `LDR` instruction.

Identifying Emulated vs. Real CPU Discrepancies with Tracing

This section describes how we use our tracing capabilities to find differences in instruction semantics between emulated and real ARM CPUs.

Generating Test Cases: We generate valid ELF binaries as inputs to our tracing platforms. We choose to use programs that contain random instructions. Specifically, each input binary contains 20 random bytes corresponding to 5 ARM instructions.

We use this randomized approach to be able to exercise emulators with uncommon instructions which are not likely to be emitted by compilers. We use more than one instruction per binary to be able to cover more instructions each time a simulation is launched for a test program.

Each test program starts with a few instructions that set the CPU state, clear registers and condition flags. By default, the programs run on the Fast Model in `svc` mode and no stack space is allocated. Thus, we use these initialization instructions to ensure that CPU mode is set to `usr` and `SP` points to the same address on both platforms. We also clear all registers to ensure that programs start from identical architectural state on both emulator and real CPU. These initialization instructions are followed by 5 random instructions. Finally, each test case ends with an `exit` system call sequence (i.e., `mov r7,#1; svc 0x0`).

Identifying Divergence Points: This phase of the PROTEUS system consumes the traces collected from QEMU and ARM Fast Model to identify and group divergent behaviors. To identify the initial point where QEMU and Fast Model traces of an input program diverge, we perform a step-by-step comparison.

The step-by-step comparison procedure is illustrated in Figure 5·9. We skip the portion of the traces which corresponds to the initialization instructions described in the previous section (Step 1) to avoid false alarms that arise from the initial state differences between QEMU and Fast Model. We walk through the remaining instruction sequence until either a difference exists in the collected trace data or the test program on QEMU terminates due to an exception. If the program terminates on QEMU or the CPU mode on Fast Models switches to a different mode than `usr`, we examine whether this exception behavior matches between QEMU and real CPU (Step 2). We perform the comparison using the CPU mode from the Fast Model and the signal received by the program upon termination on QEMU. Note

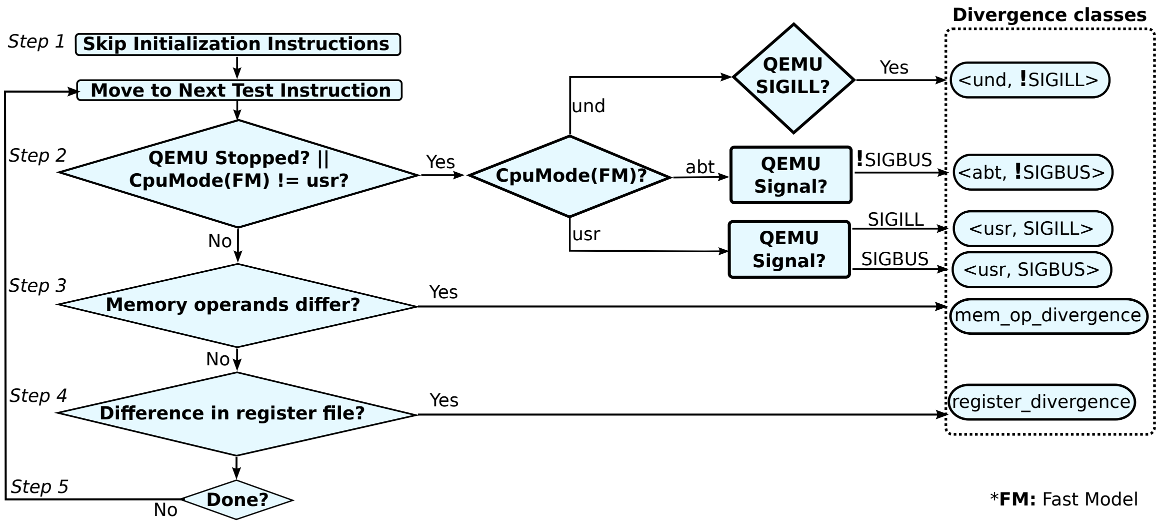


Figure 5·9: Illustration of the flow for comparing the Fast Model and QEMU traces.

that there is no exception handling or signal mechanism on Fast Models as no OS is running. Depending on this CPU mode and signal comparison, we determine whether the observed behavior falls into one of the four possible divergent types below. We use a tuple representation as `<FastModel_response, QEMU_response>` to categorize divergent behavior.

- `<und,!SIGILL>`: This group represents the cases where QEMU fails to recognize an architecturally `Undefined` instruction. If the Fast Models indicate that CPU switches to `und` mode, the expected behavior for QEMU is to deliver a `SIGILL` signal to the target program. This is because execution of an `Undefined` instruction takes the CPU into `und` mode and generates an illegal instruction exception. Thus, the cases where Fast Model switches to `und` mode while QEMU does not deliver a `SIGILL` signal is a sign of divergence.

- `<usr,SIGILL>`: This class of divergence contains cases where QEMU terminates by an illegal instruction signal (`SIGILL`) while Fast Models indicate the target instruction is valid (i.e., cpu remains in `usr` mode).

- `<abt,!SIGBUS>`: This class captures the cases where QEMU fails to recognize a data/prefetch abort and hence does not generate a bus error (i.e., deliver `SIGBUS`). Prefetch aborts are caused by failing to load a target instruction while data aborts indicate that the CPU is unable to read data from memory (e.g., due to privilage restrictions, misaligned addresses etc.) (ARM, 2018b).
- `<usr,SIGBUS>`: This divergence type represents the opposite of the previous case. Specifically, QEMU detects a bus error and delivers a `SIGBUS` to the test program while the Fast Models indicate that the memory access made by the target program is valid (i.e., cpu is not in `abt` mode).

If no exception is triggered for an instruction, we further compare the registers and memory operands within the collected trace data. We determine memory operand divergence (Step 3) if the address or the number of transferred bytes differ between QEMU and Fast Model traces. We do not treat data differences as divergence since subtle differences may exist in the initial memory states of QEMU and Fast Models. We drop cases with different memory values from further examination as the loaded data would propagate into register state and cause false positive divergence detection. Finally, if no divergence is identified in exception behavior or in memory operands, we compare the user-level registers (Step 4) to detect any register state divergence. Steps 2-4 presented in Figure 5·9 continues for the remaining random instructions in the test program.

Since `Unpredictable` instructions can cause different legitimate behaviors on any two CPU implementations, we cannot use these instructions to deterministically differentiate emulators from real systems. Thus, if a divergent instruction identified in Steps 2-4 is `Unpredictable`, we do not classify this case into any divergence group. However, an officially verified tool or a programmatic methodology to check if a given ARM instruction would generate `Unpredictable` behavior is unavailable. Thus, we use an open-source specification of ARMv7 architecture written in Standard ML (SML) and proposed by the earlier work (Fox, 2012).

Generating Detector Programs: Based on the identified divergent instruction semantics, Proteus can optionally generate a proof-of-concept program that determines whether the program is executed on an emulator or a real system. This section describes how our system generates these programs and provides insight into how a real-life malware can implement CPU semantic attacks.

Proteus generates detector programs by rewriting a set of pre-built template binaries to reconstruct the divergent behavior. The template programs simply execute the divergent instruction and check whether the resulting effect matches with QEMU or real CPU behavior. Before executing the divergent instruction, we set up the environment with the necessary changes to trigger divergent behavior and observe its effect. These changes are (1) setting the register values (`CPSR` and `R0-R12`) with the values provided from the divergence identification phase, (2) installing signal handlers for exception-related divergent behavior. We use `sigsetjmp/siglongjmp` provided by the standard C library to preserve the CPU state which would otherwise be altered by the changes we perform. Listing 5.1 illustrates an example detector program for the `<abt,!SIGBUS>` group. The current CPU state is saved with `sigsetjmp` on line 11. The register state is loaded with target values (line 13) and the divergent instruction (`LDM r7,{r4}`) is executed (line 15) to reconstruct the divergent behavior. The program determines emulation if it does not receive a `SIGBUS` after the divergent instruction accesses the misaligned memory address stored in `R4`. The original CPU state before constructing the divergent behavior is restored on either line 7 or 16. We simply build one template program for each of the six divergence groups. Depending on the divergence group, for a given divergent instruction, we pick the corresponding sample template to rewrite a new detector program.

Listing 5.1: A sample program snippet for detecting Android emulator.

```
sig_atomic_t sig_info = 0;
sigjmp_buf buf;

void catch_signals(int signum) {
  if (signum == SIGBUS)
    sig_info = 1;
  siglongjmp(buf, 1);
}
sig_atomic_t say_my_name() {
  prepare(); // register signal handlers
  if(!sigsetjmp(buf,1)) {
    // Set target registers
    asm("LDR␣␣r4,=0x00008075");
    // execute divergent instruction
    asm(".byte␣0x10,0x00,0x97,0xe8");
    siglongjmp(buf, 1);
  }
  cleanup(); // remove signal handlers
  return sig_info;
}
void main() {
  sig_atomic_t ret = say_my_name();
  (ret==1) ? printf("real␣cpu"):printf("emulator");
}
```

5.2.4 Evaluation

This section demonstrates the capabilities of PROTEUS for identifying the discrepancies of QEMU's emulated CPU from a real ARM CPU. We systematically analyze the divergences reported by PROTEUS to identify the root causes of the discrepancies. On a real smartphone and Android emulator, we demonstrate how our findings can fingerprint the underlying platform. Finally, we demonstrate the feasibility of fixing several root causes of divergences without any observable performance penalty. Overall, we seek to answer the following questions:

- Are there any observable discrepancies between an emulated and real CPU? If so, how prevalent are these differences?
- How effective are the divergences reported by PROTEUS in terms of fingerprinting real hardware and dynamic analysis platforms?
- What are the root causes of the discrepancies and can we eliminate them in QEMU without impacting its performance?

Divergence Statistics from Proteus

In order to address our first research question, we use PROTEUS to examine the instruction-level traces from 500K input test programs. Figure 5·10 shows the number of instructions executed in the test programs until a divergence occurs or QEMU stops due to an exception. The majority of the test cases (45%) finish after a single instruction only and, almost all test cases (above 94%), either diverge or cause an exception on QEMU after executing the 5 instructions in our test programs. Overall, our system analyzed over 1.06M CPU instructions. Table

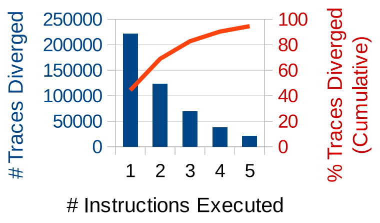


Figure 5·10: #Instructions before divergence or exception.

		QEMU Response		
		SIGILL	SIGSEGV	None
Fast Model Behavior	und	149436 (30%)	826 (0.2%)	9341 (1.9%)
	abt	11 (0.002%)	12471 (2.5%)	528 (0.1%)
	svc	0	18932 (3.8%)	14609 (2.9%)
	usr	5270 (1.1%)	176975 (35%)	0

(a) Exception behavior comparison.

`<und, !SIGILL>`	10167 (2%)
`<usr, SIGILL>`	5270 (1.1%)
`<abt, !SIGBUS>`	13010 (2.6%)
`<usr, SIGBUS>`	0
`mem_op_difference`	200 (0.05%)
`register_divergence`	12 (0.002%)

(b) Divergences by type.

Table 5.3: Divergence statistics generated by PROTEUS for 500K test cases containing 2.5M random ARM instructions. Remaining instances of 500K programs (not shown in the Table 5.3a) are (1) 83,125 (17%) cases due to `Unpredictable` instructions, (2) 27,048 (5.4%) non-divergent cases where programs finish successfully on both platforms and (3) 1216 cases that differ due to memory values. Note that we do not treat these 3 cases as divergent.

5.3 presents an overall view of the results by PROTEUS showing a comparison between QEMU and Fast Models in terms of the exception behavior (Table 5.3a) as well as extent of divergences per group (Table 5.3b).

Table 5.3a presents a summary of the cases where either QEMU terminates the program or the CPU mode changes in Fast Models. Overall, we observe two types of signals in QEMU (i.e., `SIGILL`, `SIGSEGV`) and CPU mode in Fast Models cover `und`, `abt`, `svc` and `usr` modes. `None` represents the cases where QEMU does not generate an exception. Most instances correspond to illegal instruction (`<und, SIGILL>`) and valid memory access (`<usr, SIGSEGV>`) cases in which the behavior in QEMU complies with Fast Models (i.e., not divergent). A large number of instances are *Supervisor Call* (`svc`) instructions which cover a large encoding space in ARM ISA. `svc` instructions are used to request OS services and are not a major point of interest for our work as we focus on the discrepancies that are observable in the user space. In Table 5.3a, such non-divergent cases are highlighted in gray. The remaining instances in Table 5.3a, along with the non-exception related differences (i.e., memory operand and register) are grouped into the divergence types as per the methodology described in Section 5.2.3.

Table 5.3b provides the number of instances per each divergence type. The largest number of divergences (i.e., 2.6% of 500K test programs) belong to `<abt, !SIGBUS>` group which hints that QEMU does not correctly sanitize the invalid memory references that cause data/prefetch aborts in CPU. Proteus also finds a large number of instructions that are recognized as architecturally `Undefined` only by the Fast Models (i.e., `<und, !SIGILL>` group). These point to cases where QEMU does not properly validate the encoding before treating the instruction as architecturally valid. We also find a large number of instructions which are detected as illegal only by QEMU, executing without raising an illegal instruction exception on the Fast Model (i.e., `<usr, SIGILL>` group). Proteus also finds a smaller number of cases (i.e., 0.05%) with divergent register update or memory operation which correspond to `register_divergence` and `mem_op_difference` groups in Table 5.3b, respectively. These examples hint at cases where the implementation of a valid instruction contains potential errors in QEMU, causing a different register or memory state than on a real CPU. Overall, despite the significant testing of QEMU, we observe that there are still many divergences where QEMU does not implement the ARM ISA faithfully.

Root Cause Analysis

While the Proteus system can identify large numbers of discrepancies between real and emulated ARM CPU, it does not pinpoint the root causes in QEMU that lead to a different behavior than ground truth (i.e., Fast Model behavior). This section presents our findings from an analysis of root causes of divergent behavior in QEMU.

This analysis gives us, compared to large number of divergences identified, a smaller set of unique errors in QEMU that lead to divergence on a wide set of programs (Table 5.3b). Analyzing the root causes also allows us to pinpoint implementation flaws and devise fixes.

In our analysis, for a divergence group, we first identify common occurrences in the bit fields *[27:20]* of a divergent 32-bit instruction encoding. In the ARM architecture, these bits contain opcodes that are checked while decoding the instruction on QEMU and real CPU. We identify the instructions with the most commonly occuring opcodes to (1) consult the ISA specification to check how these instruction should be decoded and (2) check how QEMU processes these instruction. We determine the root cause of the discrepancy by manually analyzing QEMU's control flow while executing a sample of these instructions. Once we examine the source of discrepancy (e.g., a missing check, an unimplemented feature of QEMU), we remove all possible encodings that stem from the same root cause from our statistics to find other unique instances of errors in QEMU.

Through this iterative procedure, we identified several important classes of flaws in QEMU that result in a different instruction-level behavior than a real CPU. We discuss some of our findings in the following paragraphs.

Incomplete Sanitization for Undefined Instructions: We discover that QEMU does not correctly generate illegal instruction exception for a set of `Undefined` instruction encodings. These cases are identified from the `<und, !SIGILL>` group provided by PROTEUS. Thus, a malware can achieve evasion simply by executing one of these instructions and ceasing malicious activity if no illegal instruction exception is generated.

We find that this particular group of divergences arises as QEMU relaxes the number of checks performed on the encoding while decoding the instructions. For instance,

Instruction Encoding (`cond` $\in [0, 0xE]$)	**Divergent Condition**	**QEMU Behavior**	**Real CPU Behavior**	**#Cases**
`cond:4|0001|op:4|*:12|1001|*:4`	op = 1,2,3,5,6,7	SWP Inst.	`Undefined`	715
`cond:4|1100010|*:9|101|*:1|op:4|*:4`	op != 1,3	64-bit VMOV	`Undefined`	424
`cond:4|11|op:6|*:20`	op = 1,2	VFP Store	`Undefined`	51
`cond:4|11101|*:2|0|*:8|1011|*:1|op:2|1|*:4`	op = 2,3	VDUP Inst.	`Undefined`	3
`cond:4|110|op:5|*:8|101|*:9`	op != 4,5,8-25, 28,29	VFP Store	`Undefined`	2

Table 5.4: Several `Undefined` instruction encodings that are treated as valid instructions by QEMU. ":X" notation represents the bit length of a field while "*" represents that the field can be filled with any value (i.e., 0 or 1).

the ARM ISA defines a set of opcodes for which the synchronization instructions (e.g., `SWP`, `LDREX`) are `Undefined`, and thus should generate an illegal instruction exception. However, QEMU does not check against these invalid opcodes while decoding the synchronization instructions, causing a set of `Undefined` encodings to be treated as a valid `SWP` instruction. In fact, we identified 715 divergent test cases which are caused by this missing invalid opcode check for the `SWP` instruction. In Table 5.4, we provide the encoding and the conditions that cause divergent behavior for this `SWP` instruction example as well as other similar errors in QEMU that we have identified.

During our root cause analysis, we find that a large portion of the instances in `<und, !SIGILL>` group (87%) are due to instructions accessing the co-processors with ids 1 and 2. These co-processors correspond to FPA11 floating-point processor that existed in earlier variants of the ARM architecture while newer architectures (¿ARMv5) use co-processor 10 for floating point (VFP) and 11 for vector processing (SIMD). While accesses to co-processors 1 and 2 are `Undefined` on a real CPU, QEMU still supports emulation of these co-processors (Bambrough, 1999). Thus, these instructions generate an illegal instruction exception only on the real CPU.

Misaligned Memory Access Checks: As hinted by PROTEUS with the large number of instances in the `<abt, !SIGBUS>` group in Table 5.3b, we identify that

QEMU does not enforce memory alignment requirements (e.g., alignment at word boundaries) for the ARM instructions that do not support misaligned memory accesses. The data aborts caused by such misaligned accesses would take the CPU into `abt` mode and the program is expected to be signalled with `SIGBUS` to notify that the memory subsystem cannot handle the request. Due to missing alignment checks in QEMU, a malware can easily fingerprint emulation by generating a memory reference with a misaligned address and observing whether the operation succeeds (i.e., in QEMU) or fails (i.e., on a real system).

The ARMv7 implementations can support misaligned accesses for the load/store instructions that access a single word (e.g., `LDR`, `STR`), a half-word (e.g., `LDRH`, `STRH`) or only a byte of data (e.g., `LDRB`, `STRB`). However, other instructions that perform multiple loads/stores (e.g., `LDM`, `STM`) or memory-register swaps for synchronization (e.g., `SWP`, `LDREX`, `STREX`) require proper alignment of the data being referenced. The alignment requirement can be word, half-word or double-word depending on the size of data being accessed by the instruction.

Updates to Execution State Bits: By analyzing the divergent instructions reported by PROTEUS within the `register_divergence` group, we identified another important root cause in QEMU due to masking out of the execution state bits during a status register update. Specifically, we analyzed the cases where execution state bits within `CPSR` differ after an `MSR` (move to special registers) instruction. Execution state bits in `CPSR` determine the current instruction set (e.g., ARM, Thumb, Jazelle) and the endianness for loads and stores. While `MSR` system instructions allow to update `CPSR`, writes to execution state bits are not allowed with the only exception being the endianness bit (`CPSR.E`). The ARM ISA specifies that "`CPSR.E` bit is writable from any mode using an `MSR` instruction" (ARM, 2018b). However, since updates on the `CPSR.E` bit by an `MSR` instruction are ignored in current QEMU, software can easily fingerprint the emulation by simply trying to flip this bit (e.g., using `MSR CPSR_x, 0x200` instruction) and checking whether the endianness has been succesfully changed.

Observations from other statistics: Our initial investigations on `<usr, SIGILL>` and `mem_op_divergence` groups did not reveal any further root causes as above. We find that the majority of the divergent cases in `mem_op_divergence` group (¿97%) are due to VFP/SIMD instructions effecting the extension registers. Our current work focuses on the user-mode general purpose registers only. During analysis on `<usr, SIGILL>` group, we identified divergences due to `Unpredictable` instructions. This issue is due to the incomplete SML model (Fox, 2012) which misses some `Unpredictable` instructions in our test cases (Figure 5·7). For instance, we find that 761 divergence cases in `<usr, SIGILL>` group are due to `Unpredictable` encodings of a `PLD` (i.e., preload data) instruction, which behave as a `NOP` in Fast Model but generate an illegal instruction exception in QEMU.

Demonstration with Real Smartphones and the SDK Emulator

In this section, we address our second research question on evaluating the effectiveness of the divergences found by PROTEUS for real-world emulation detection. To tackle this objective, we evaluate the divergences identified by Proteus on a physical mobile platform and Android emulator. We use Nexus 5 (ARMv7) and Nexus 6P (ARMv8) smartphones as our real hardware test-beds and use the full-system emulator from the Android SDK. We choose the SDK emulator as it has been a popular base for Android dynamic analysis frameworks (Yan and Yin, 2012; Oberheide and Miller, 2012; Tam et al., 2015).

Evaluating Unsanitized `Undefined` Encodings: We use the detection binaries generated by PROTEUS to evaluate the `Undefined` instructions that are incompletely sanitized in QEMU (i.e., `<und, !SIGILL>` group). These cases are expected to generate an illegal instruction exception only on a real CPU.

We find that the SDK's copy of QEMU does not incorporate the FPA11 floating point co-processor emulation which is supported in our version of QEMU and accessed by the instructions that use co-processors 1 and 2. Thus, these instructions are `Undefined` in SDK emulator as well and we cannot successfully distinguish the emulator from the real hardware. As discussed previously, FPA11 instructions account for 87% of the cases in `<und, !SIGILL>` group. However, we can successfully fingerprint the SDK emulator using all the other divergent `Undefined` instructions. Specifically, all the encodings described in Table 5.4 can deterministically distinguish between SDK emulator and Nexus 5. The detector programs simply register a set of signal handlers and detect the SDK emulator if the program does not receive `SIGILL` upon executing the divergent `Undefined` instruction.

Listing 5.2: PoC for emulator detection by flipping endianness bit.

```
/* Put some known data into memory */
int *ptr = calloc(1,sizeof(int));
ptr[0] = 0x12345678;
asm("mov r8,%0" : : "r"(ptr));

/* Read ptr[0] with CPSR.E set to 1 */
asm("msr CPSR_x, #0x200\n\t");
asm("ldr r4,[r8]\n\t");
asm("msr CPSR_x, #0x000\n\t");

asm("mov %0, r4" : "=r"(val) : : );
printf("0x%08X\n", val);
```

Listing 5.3: PoC for emulator detection by misaligned memory read.

```
/* Put some known data into memory */
int *ptr = calloc(1,sizeof(int));
ptr[0] = 0x12345678;

//Shift address to a non-word boundary
ptr = (int*)((char*)ptr + 0x1);

//Try to read from misaligned address
asm("mov r3,%0" : : "r"(ptr));
asm("LDM r3,{%0}": "=r"(val) : : );

printf("0x%08X\n", val);
```

Evaluating Missing Alignment Checks and Endianness Support: We also show that we can successfully detect the SDK emulator by leveraging the ignored endianness bit updates as well as the missing memory address alignment checks. Listing 5.2 provides a proof-of-concept (PoC) code sample that fingerprints emulation by flipping the endianness bit in the CPU (i.e., `CPSR.E`) and performing a load operation on a known data value to determine whether the endianness has been changed. Executing this code snippet on a real hardware (i.e., Nexus 5 in our case) reads the array value as `0x78563412` instead of `0x12345678` as the `CPSR.E` bit is set to switch from little-endian to big-endian operation for data accesses. However, since the `CPSR.E` bit update is ignored in QEMU, the `LDR` instruction reads the array element into `R4` as `0x12345678` on the SDK emulator. Thus, a malware can easily fingerprint emulation by simply checking the value of target register (i.e., `R4` in this example). Similarly, Listing 5.3 illustrates how the missing alignments checks in QEMU can be leveraged to fingerprint emulation. We shift the word-aligned `ptr` pointer by one byte to create a misaligned reference address. Reading from this misaligned pointer with an `LDM` (i.e., load multiple) instruction causes a bus error (program receives `SIGBUS`) on a real hardware while it succesfully reads the high-order part (`0x123456`) of the target address on the SDK emulator. Thus, a malware can simply determine emulation depending on whether a `SIGBUS` signal is received (i.e., on a real system) upon intentionally causing a misaligned memory access.

Evaluation on a ARMv8 CPU: The 64-bit ARMv8 architecture, which is used in recent smartphones, is compatible with ARMv7. Thus, the CPU semantic attacks we demonstrate in this work also apply to devices powered with ARMv8 CPUs (e.g., Nexus 6P). We evaluated PoC detectors for each root cause we discovered (i.e., Table 5.4, Listings 5.2 and 5.3) on a Nexus 6P smartphone and successfully distinguished this device from the SDK emulator as well.

Improving the Fidelity of QEMU

With the capabilities of Proteus for identifying and classifying divergences in instruction-level behavior, in this section, we show the feasibility of eliminating the sources of discrepancies to improve QEMU's fidelity.

We have modified the QEMU source code of the SDK emulator to eliminate the top 3 detection methods in Table 5.4 based on incomplete sanitization of opcodes for `Undefined` encodings. Specifically, based on the ARM ISA specification (ARM, 2018b), we fixed the decoding logic of QEMU to verify all opcode fields for these 3 cases and trigger an illegal instruction exception for the `Undefined` encodings. These fixes eliminated 1190 divergent cases in Table 5.4. Using various CPU benchmarks from MiBench suite (Guthaus et al., 2001), in Figure

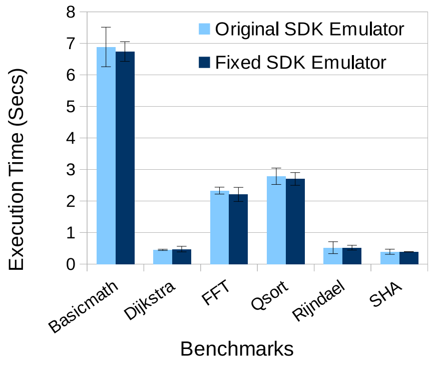


Figure 5·11: Overhead evaluation of fidelity enhancements.

5·11, we verified that the minimal extra code needed to perform additional opcode checks does not introduce any measurable performance overhead. We acknowledge, however, that addressing the alignment check and endianness support in QEMU will require more comprehensive changes than the missing opcode checks for `Undefined` encodings.

Chapter 6

Conclusion and Open Problems

This thesis has presented techniques to improve sustained performance under thermal limitations of contemporary mobile devices. We have presented thermally-efficient runtime strategies that leverage insights from real-world mobile applications as well as providing software frameworks to aid systematic study of such applications. In this section, we briefly discuss several future directions for these techniques. We also summarize the distinguishing aspects of this work.

6.1 Future Research Directions and Open Problems

As the computational power needed to support the emerging field of applications (e.g., AI/ML, healthcare) increases, energy and thermal restrictions of mobile devices will be even more crucial. Therefore, innovations across all layers of the computing stack is necessary. For instance, use of domain-specific hardware accelerators to efficiently execute common daily use cases can drastically reduce power while improving performance (e.g., for Web browsing (Zhu and Reddi, 2014) or machine learning (Chen et al., 2014)). Similarly, more synergistic design of applications or system software together with the hardware architecture can unlock orders of magnitude improvements in energy efficiency and performance (Zhu et al., 2018; Belay et al., 2014; Belay et al., 2012). In this section, we describe several specific future steps for the work presented in this thesis. In Section 6.1.1, we describe several future directions to improve the mobile system efficiency through advancements in system software. Section 6.1.2

discusses the future work for record and replay of mobile applications.

6.1.1 Runtime Management in Mobile Systems

This thesis has presented runtime techniques to provide extended durations of sustained QoS. We have proposed to tune the short-term QoS under thermal limitations to provide "just enough" performance for the user needs. Through real-system evaluations, we have demonstrated significantly longer durations of sustained QoS using our techniques. However, there are several opportunities to advance our runtime policies and extend the scope of our work.

Current schedulers in state-of-the-art heterogeneous mobile platforms take actions based on coarse-grained utilization metrics. In our work (Sahin and Coskun, 2016b), we have demonstrated the benefits of guiding scheduling decisions based on the criticality of thread in terms of QoS in mobile applications. However, identifying the QoS critical threads in a systematic and practical manner remains an important open problem. A promising approach to address this problem could be to provide runtime management with information from the application or the Android framework. Prior work (Zhang et al., 2013; Ravindranath et al., 2012) has proposed techniques to extract various critical computations within the application as well as the framework that relate to user perceived performance. Such identification of critical computations can be provided as an input to runtime management to guide scheduling decisions.

The runtime policies we have presented in this thesis focus primarily on the CPU subsystem. Current mobile SoCs employ increasing number of on-chip accelerators (e.g., a GPU), which can also substantially contribute to power consumption (Halpern et al., 2016; Hill and Reddi, 2019). Thus, further improvements in sustained QoS durations can be achieved by regulating the power consumption of other IP blocks than the CPU to deliver "just enough" performance. However, managing the power and performance tradeoffs across multiple IP blocks is a challenging problem due to the need to consider the tight interactions between IPs (Nachiappan et al., 2015).

6.1.2 Record and Replay for Android

The record and replay framework we have proposed for Android, RandR (Sahin et al., 2019), provides cross-platform reproducibility of multiple non-deterministic input sources. Our current implementation of RandR, however, can be improved along several dimensions to provide higher replay fidelity and generality.

RandR currently does not support record and replay of various other non-deterministic input sources such as GPS or other sensor data. This limitation can be addressed by identifying the additional instrumentation points in the Android framework. RandR can be provided with the additional instrumentation points as well as the necessary logic for record and replay to handle other input sources.

RandR only supports cross-platform replayability for the applications that use the Android's UI toolkit (i.e., widgets). However, various applications (i.e., primarily games) render their UI screen natively without relying on the UI components provided by Android. This problem can be addressed to some extent with learning-based approaches that automatically analyze user-visible screen states and mimic the recorded user interactions (Hu et al., 2018).

RandR also does not guarantee any scheduling order among threads. Thus, RandR may not be able to accurately reproduce any application behavior that depend on the order of execution between threads (e.g., race conditions). Prior work has enforced such ordering constraints with modifications to the kernel (Veeraraghavan et al., 2012). However, such modifications would change the nature of RandR as they would necessitate elevated privileges on the target device.

6.2 Summary of Major Contributions

As the mobile SoC designs push the performance envelope to serve our complex daily applications with high performance, power and thermal limitations have became a major roadblock. Increased power densities cause thermal violations and lead to unsustainability of the QoS levels delivered to users. This thesis has claimed that systematic understanding of real-world mobile applications and leveraging insights from their characteristics allow to exploit unique tradeoffs between temperature and QoS, and maximize sustained performance. To this end, we proposed 1) runtime management techniques that provide thermally-efficient and application-aware QoS management for mobile applications; 2) software frameworks to enable systematic and reproducible experiments with real-world mobile applications.

This thesis first experimentally shows that existing thermal management strategies in mobile devices that greedily boost performance can result in significant QoS loss over extended durations of application use (e.g., as in gaming). We have argued for *QoS-centric thermal management* that is optimized for *"just enough"*, as opposed to the maximum, performance needed to satisfy user demands.

We have proposed thermally-efficient QoS control techniques for both homogeneous and heterogeneous multi-core CPUs. Our closed-loop QoS controller with thermally-efficient DVFS state scheduling technique reduces peak temperatures by exploiting thermal time constants while precisely delivering QoS targets. Such DVFS-based policies can be broadly applied to both homogeneous and heterogeneous CPU platforms. However, leveraging the unique power/performance tradeoffs offered by heterogeneous CPUs (e.g., big.LITTLE) via efficient scheduling can provide further room for optimization. To this end, we have proposed a thermally-efficient QoS control framework, QScale, for heterogeneous platforms. QScale exploits low TLP in mobile applications along with considering the QoS criticality of individual threads to reduce the load on power-hungry cores while meeting QoS targets. QScale strategically activates the power-hungry cores that provide the most thermally-efficient operation by monitoring CPU-GPU thermal couplings at runtime. Overall, our real-system implementation and evaluation of QScale has shown up to 8x longer durations of sustained QoS.

Our thermally-efficient runtime techniques described above meets a target QoS constraint while minimizing temperatures, A related important question is how to determine the target QoS levels for mobile applications. We have addressed this problem with a runtime QoS management framework, Maestro. Maestro identifies an application's compute behavior in runtime and manages 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. For the cases of continuous throughput-oriented computations with high power, Maestro proactively trades off QoS to enable sustained QoS over the use. Through real-life implementation and evaluation, we have shown Maestro's ability to adapt to application behavior and autonomously manage QoS to improve the durations of sustained performance by 41% to 6.7x.

Our thermally-efficient runtime management strategies described above derives insights from the real-world behavior of mobile applications (e.g., QoS contraints, bursty and throughput-oriented computations). Systematically exploring real-world behavior of mobile applications, however, has proven challenging. This is due to non-deterministic behavior of mobile applications whose execution vastly vary due to multiple sources of input (e.g., UI, network). This thesis provides software frameworks for systematic evaluation of real-world mobile applications.

We have presented the RandR system for record and replay of Android applications to enable reproducibility of their executions while also eliminating the deployment challenges of existing toos. RandR captures the inputs to an application by hooking into a specific set of instrumentation points in the Android framework and native libraries. By performing the instrumentation within an application's own sandbox, RandR runs on unmodified devices and works with real-world closed-source applications with only minimal modifications to an application's bytecode. RandR captures UI and random number inputs by hooking into a set of Java APIs in the Android framework and provides widget-sensitive cross-platform replay capabilities. RandR also reproduces network traffic in applications by intercepting the random inputs in the TLS protocol as well as the system call wrappers methods in libc. We have implemented a prototype of RandR and demonstrated its accurate record and replay capabilities using 10 real-world Android applications both qualitatively (i.e., via visual comparison) and quantitatively (i.e., via the similarity between the sets of executed methods).

We have also studied the malicious non-determinism in Android applications that seek to evade emulated dynamic analysis sandboxes. We have presented the first systematic study of differences in instruction-level behavior of emulated and real ARM CPUs that power the vast majority of Android devices. We have proposed the Proteus system to automatically analyze detailed instruction-level traces collected from QEMU and accurate software models of ARM CPUs, and identify discrepancies in the instruction-level behavior. Our evaluation has revealed several major root causes for instruction-level discrepancies in QEMU. We have also demonstrated the feasibility of enhancing the fidelity of QEMU by fixing the root causes of divergences without any performance impact.

Overall, we have shown that the thermally-efficient runtime management strategies we have presented in this work can improve the durations of sustained QoS by up to 8x. We have also proposed a technique to autonomously manage QoS tradeoffs to achieve sustainable performance in real-world mobile applications. Finally, in order to facilitate the mobile system research on real-life systems (i.e., as in this thesis), we have presented software frameworks to aid systematic study of real-world mobile application behavior.

References

AOSP (2019). Performance management. `https://source.android.com/devices/tech/power/performance`. [Online].

Aquarium (2018). Aquarium. `http://webglsamples.org/aquarium/aquarium.html`. [Online].

ARM (2013). big.LITTLE Technology. `https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf`. [Online].

ARM (2018a). ARM Fast Models. `https://developer.arm.com/products/system-design/fast-models`. [Online].

ARM (2018b). ARMv7-A/R Architecture Reference Manual. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html`. [Online].

Ayoub, R. Z. et al. (2011). Os-level power minimization under tight performance constraints in general purpose systems. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design (ISLPED).*

Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., and Vigna, G. (2010). Efficient detection of split personalities in malware. In *Networked and Distributed Systems Security (NDSS).*

Bambrough, S. (1999). NetWinder Floating Point Notes. `http://netwinder.osuosl.org/users/s/scottb/public_html/notes/FP-Notes-all.html`. [Online].

Bartolini, A., Cacciari, M., Tilli, A., and Benini, L. (2011). A distributed and self-calibrating model-predictive controller for energy and thermal management of high-performance multicores. In *Design, Automation Test in Europe (DATE)*, pages 1–6.

Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. (2012). Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 335–348, Berkeley, CA, USA. USENIX Association.

Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. (2014). Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 49–65, Berkeley, CA, USA. USENIX Association.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Bordoni, L., Conti, M., and Spolaor, R. (2017). Mirage: Toward a stealthier and modular malware analysis sandbox for android. In *Computer Security – ESORICS*.

Branco, R. R., Barbosa, G. N., and Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. In *BlackHat*.

Brodowski, D. (2012). Android cpu governors. `https://android.googlesource.com/kernel/common/+/a7827a2a60218b25f222b54f77ed38f57aebe08b/Documentation/cpu-freq/governors.txt`. [Online].

CamanJS (2018). Camanjs image editor. `http://camanjs.com/`. [Online].

Chantem, T., Hu, X. S., and Dick, R. P. (2009). Online work maximization under a peak temperature constraint. In *International Symposium on Low-Power Electronic Devices (ISLPED)*, pages 105–110.

Chaturvedi, V., Huang, H., and Quan, G. (2010). Leakage aware scheduling on maximum temperature minimization for periodic hard real-time systems. In *IEEE 10th International Conference on Computer and Information Technology (CIT)*.

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, New York, NY, USA. ACM.

Chung, H. (2012). Heterogeneous multi-processing. `https://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf`. [Online].

Continella, A., Fratantonio, Y., Lindorfer, M., Puccetti, A., Zand, A., Kruegel, C., and Vigna, G. (2017). Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *24th Annual Network and Distributed System Security Symposium (NDSS)*.

Costamagna, V. and Zheng, C. (2016). Artdroid: A virtual-method hooking framework on android art runtime. In *IMPS@ESSoS*.

Cunningham, A. (2013). When benchmarks aren't enough: Cpu performance in the nexus 5. [online] `https://arstechnica.com/gadgets/2013/11/when-benchmarks-arent-enough-cpu-performance-in-the-nexus-5/`.

Davis, G. (2017). Grabos Malware. `https://securingtomorrow.mcafee.com/consumer/consumer-threat-notices/grabos-malware/`. [Online].

Egilmez, B., Memik, G., Ogrenci-Memik, S., and Ergin, O. (2015). User-specific skin temperature-aware dvfs for smartphones. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1217–1220.

EIA/JEDEC (2016). Failure mechanisms and models for semiconductor devices, JEDEC publication JEP122C. `https://www.jedec.org/standards-documents/docs/jep-122e`.

EMMA (2006). Emma. `http://emma.sourceforge.net`.

Emurian, L., Raghavan, A., Shao, L., Rosen, J. M., Papaefthymiou, M., Pipe, K., Wenisch, T. F., and Martin, M. (2014). Pitfalls of accurately benchmarking thermally adaptive chips. *Power (W)*, 5:10.

Endo, Y., Wang, Z., Chen, J. B., and Seltzer, M. (1996). Using latency to evaluate interactive system performance. *SIGOPS Operating Systems Review*.

Espresso, A. (2019). Espresso. `https://developer.android.com/training/testing/espresso/`. Accessed: 2018-10-29.

Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., and Estrin, D. (2010). Diversity in smartphone usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 179–194, New York, NY, USA. ACM.

Fazzini, M., Freitas, E. N. D. A., Choudhary, S. R., and Orso, A. (2017). Barista: A technique for recording, encoding, and running platform independent android tests. In *International Conference on Software Testing, Verification and Validation (ICST)*.

Fox, A. (2012). Directions in isa specification. In Beringer, L. and Felty, A., editors, *Interactive Theorem Proving (ITP). Lecture Notes in Computer Science*, volume 7406, pages 338–344, Berlin, Heidelberg. Springer.

Gao, C., Gutierrez, A., Dreslinski, R. G., Mudge, T., Flautner, K., and Blake, G. (2014). A study of thread level parallelism on mobile devices. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 126–127.

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

Gomez, L., Neamtiu, I., Azim, T., and Millstein, T. (2013). Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81, Piscataway, NJ, USA. IEEE Press.

Goo (2018). Pearl Boy. `https://www.chromeexperiments.com/experiment/pearl-boy`.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization (IISWC)*.

Gutierrez, A., Dreslinski, R. G., Wenisch, T. F., Mudge, T., Saidi, A., Emmons, C., and Paver, N. (2011). Full-system analysis and characterization of interactive smartphone applications. In *International Symposium on Workload Characterization (IISWC)*, pages 81–90. IEEE.

Halpern, M., Zhu, Y., Peri, R., and Reddi, V. J. (2015). Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE.

Halpern, M., Zhu, Y., and Reddi, V. (2016). 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)*.

Hardkernel (2017). Odroid-XU3 Mobile Development Board. `https://www.hardkernel.com/ko/tag/odroid-xu3/`.

Hashemi, M., Marr, D., Carmean, D., and Patt, Y. (2015). Efficient execution of bursty applications. *IEEE Computer Architecture Letters*, PP(99):1–1.

Hellerstein, J. L. et al. (2004). *Feedback control of computing systems*. John Wiley & Sons.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17.

Hill, M. and Reddi, V. J. (2019). Gables: A roofline model for mobile socs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 317–330. IEEE.

Ho, J. and Frumusanu, A. (2014). Revisiting SHIELD Tablet: Gaming Battery Life and Temperatures. [online] Available: `https://www.anandtech.com/show/8329/revisiting-shield-tablet-gaming-ux-and-battery-life`.

Hoffmann, H. (2015). Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 198–214.

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

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

Hsiu, P.-C., Tseng, P.-H., Chen, W.-M., Pan, C.-C., and Kuo, T.-W. (2016). User-centric scheduling and governing on mobile devices with big.little processors. *ACM Transactions on Embedded Computing Systems*, 15(1):17:1–17:22.

Hu, G., Zhu, L., and Yang, J. (2018). Appflow: Using machine learning to synthesize robust, reusable ui tests. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 269–282, New York, NY, USA. ACM.

Hu, Y., Azim, T., and Neamtiu, I. (2015). Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 349–366, New York, NY, USA. ACM.

Huang, Y., Zha, Z., Chen, M., and Zhang, L. (2014). Moby: A mobile benchmark suite for architectural simulators. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 45–54.

Isci, C., Contreras, G., and Martonosi, M. (2006). Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 359–370. IEEE Computer Society.

Jing, Y., Zhao, Z., Ahn, G.-J., and Hu, H. (2014). Morpheus: Automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, pages 216–225. ACM.

Kadjo, D., Ayoub, R., Kishinevsky, M., and Gratz, P. V. (2015). A control-theoretic approach for energy efficient cpu-gpu subsystem in mobile platforms. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, pages 62:1–62:6.

Kadjo, D. et al. (2014). Towards platform level power management in mobile systems. In *27th IEEE International System-on-Chip Conference (SOCC)*, pages 146–151. IEEE.

Khdr, H., Pagani, S., Shafique, M., and Henkel, J. (2015). Thermal constrained resource management for mixed ilp-tlp workloads in dark silicon chips. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*.

Khdr, H., Pagani, S., Sousa, E., Lari, V., Pathania, A., Hannig, F., Shafique, M., Teich, J., and Henkel, J. (2017). Power density-aware resource management for heterogeneous tiled multicores. *IEEE Transactions on Computers*, 66(3):488–501.

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

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

Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. (2003). Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 81–, Washington, DC, USA. IEEE Computer Society.

Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. (2004). Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *International Symposium on Computer Architecture (ISCA)*.

Lanier, T. (2017). ARM Cortex-A15 processor. `https://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf`.

Lee, T. (2014). Battery life most important feature in a smartphone [survey]. `https://www.ubergizmo.com/2014/05/battery-life-most-important-feature-in-a-smartphone-survey/`.

Li, X., Chen, G., and Wen, W. (2017). Energy-efficient execution for repetitive app usages on big.little architectures. In *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 44:1–44:6, New York, NY, USA.

Lim, C. H., Daasch, W. R., and Cai, G. (2002). A thermal-aware superscalar microprocessor. In *Proceedings. International Symposium on Quality Electronic Design (ISQED)*, pages 517–522.

Lindorfer, M., Kolbitsch, C., and Milani Comparetti, P. (2011). Detecting environment-sensitive malware. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, pages 338–357. Springer-Verlag.

Liu, L., Gu, Y., Li, Q., and Su, P. (2017). Realdroid: Large-scale evasive malware detection on "real devices". In *26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8.

Lo, D., Cheng, L., Govindaraju, R., Barroso, L. A., and Kozyrakis, C. (2014). Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*.

Lody (2018). Andhook. `https://github.com/asLody/AndHook`.

Martignoni, L., McCamant, S., Poosankam, P., Song, D., and Maniatis, P. (2012). Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 337–348, New York, NY, USA. ACM.

Martignoni, L., Paleari, R., Roglia, G. F., and Bruschi, D. (2009). Testing cpu emulators. In *International Symposium on Software Testing and Analysis (ISSTA)*.

Mudge, T. (2001). Power: A first-class architectural design constraint. *Computer*, 34(4):52–58.

Muller, M. (2014). Power constraints: From sensors to servers. [online] `http://www.hotchips.org/wp-content/uploads/hc_archives/hc26/HC26-11-day1-epub/HC26.11-1-Z-Keynote1-Power-epub/HC26.11.190-Keynote1-Power-Muller-ARM-MM-v8.pdf`.

Muthukaruppan, T. S., Pricopi, M., Venkataramani, V., Mitra, T., and Vishin, S. (2013). Hierarchical power management for asymmetric multi-core in dark silicon era. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*, pages 174:1–174:9, New York, NY, USA. ACM.

Nachiappan, N. C., Zhang, H., Ryoo, J., Soundararajan, N., Sivasubramaniam, A., Kandemir, M. T., Iyer, R., and Das, C. R. (2015). Vip: Virtualizing ip chains on handheld platforms. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 655–667, New York, NY, USA. ACM.

Oberheide, J. and Miller, C. (2012). Dissecting the android bouncer. `https://jon.oberheide.org/files/summercon12-bouncer.pdf`.

O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., and Partush, N. (2017). Engineering record and replay for deployability. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 377–389, Santa Clara, CA. USENIX Association.

Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. (2009). A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies (WOOT)*.

Pandiyan, D., Lee, S., and Wu, C. (2013). Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - mobilebench. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 133–142.

Park, J.-G., Hsieh, C.-Y., Dutt, N., and Lim, S.-S. (2016). Co-cap: Energy-efficient cooperative cpu-gpu frequency capping for mobile games. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1717–1723, New York, NY, USA. ACM.

Pathania, A., Irimiea, A. E., Prakash, A., and Mitra, T. (2015). Power-performance modelling of mobile gaming workloads on heterogeneous mpsocs. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6.

Pathania, A., Jiao, Q., Prakash, A., and Mitra, T. (2014). Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*, pages 40:1–40:6, New York, NY, USA. ACM.

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

Patil, H., Pereira, C., Stallcup, M., Lueck, G., and Cownie, J. (2010). Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 2–11, New York, NY, USA. ACM.

Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., and Ioannidis, S. (2014). Rage against the virtual machine: Hindering dynamic analysis of android malware. In *European Workshop on System Security (EuroSec)*, pages 5:1–5:6.

Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. (2014). Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Network and Distributed System Security Symposium (NDSS)*.

Pozo, R. and Miller, B. (2000). Scimark 2.0. *URL: http://math. nist. gov/scimark2*.

Prakash, A., Amrouch, H., Shafique, M., Mitra, T., and Henkel, J. (2016). Improving mobile gaming performance through cooperative cpu-gpu thermal management. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)*, pages 47:1–47:6, New York, NY, USA. ACM.

Qin, Z., Tang, Y., Novak, E., and Li, Q. (2016). Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE, pages 571–582, New York, NY, USA. ACM.

Rao, K., Wang, J., Yalamanchili, S., Wardi, Y., and Handong, Y. (2017). Application-specific performance-aware energy optimization on android mobile devices. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 169–180.

Ravindranath, L., Padhye, J., Agarwal, S., Mahajan, R., Obermiller, I., and Shayandeh, S. (2012). Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 107–120, Berkeley, CA, USA. USENIX Association.

RK700 (2018). Yahfa. `https://github.com/rk700/YAHFA`.

RobotiumTech (2019). Robotium. `https://github.com/RobotiumTech/robotium`. Accessed: 2019-02-15.

Sahin, O., Aliyeva, A., Mathavan, H., Coskun, A., and Egele, M. (2019). Towards practical record and replay for mobile applications. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC'19, pages 230:1–230:2, New York, NY, USA. ACM.

Sahin, O. and Coskun, A. K. (2015). On the impacts of greedy thermal management in mobile devices. *IEEE Embedded Systems Letters*, 7(2):55–58.

Sahin, O. and Coskun, A. K. (2016a). Providing sustainable performance in thermally constrained mobile devices. In *Proceedings of the 14th ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 72–77, New York, NY, USA. ACM.

Sahin, O. and Coskun, A. K. (2016b). Qscale: Thermally-efficient qos management on heterogeneous mobile platforms. In *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD)*, pages 125:1–125:8, New York, NY, USA. ACM.

Sahin, O., Coskun, A. K., and Egele, M. (2018). Proteus: Detecting android emulators from instruction-level profiles. In Bailey, M., Holz, T., Stamatogiannakis, M., and Ioannidis, S., editors, *Research in Attacks, Intrusions, and Defenses*, pages 3–24, Cham. Springer International Publishing.

Sahin, O., Thiele, L., and Coskun, A. K. (2018). Maestro: Autonomous qos management for mobile applications under thermal constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 1–1.

Sahin, O., Varghese, P., and Coskun, A. (2015). 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.

Seo, W., Im, D., Choi, J., and Huh, J. (2015). 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.

Shafique, M., Gnad, D., Garg, S., and Henkel, J. (2015). Variability-aware dark silicon management in on-chip many-core systems. In *Design, Automation and Test in Europe (DATE)*, pages 387–392.

Sharifi, S., Coskun, A., and Rosing, T. (2010). Hybrid dynamic energy and thermal management in heterogeneous embedded multiprocessor socs. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 873–878.

Sheepeuh (2018). Rain. `https://codepen.io/Sheepeuh/pen/cFazd`.

Shi, H., Alwabel, A., and Mirkovic, J. (2014). Cardinal pill testing of system virtual machines. In *23rd USENIX Conference on Security Symposium*.

Singla, G., Kaur, G., Unver, A. K., and Ogras, U. Y. (2015). Predictive dynamic thermal and power management for heterogeneous mobile platforms. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 960–965, San Jose, CA, USA. EDA Consortium.

Skadron, K. et al. (2002). Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, Washington, DC, USA.

Skadron, K., Stan, M. R., Huang, W., Velusamy, S., Sankaranarayanan, K., and Tarjan, D. (2003). Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, New York, NY, USA. ACM.

Snapdragon MSM8974 MDP. (2014). [online] https://developer.qualcomm.com/hardware/mdp-mobile-development-platform.

Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *Network and Distributed System Security Symposium (NDSS)*.

Thuxnder (2015). QEMU emulation detection. `https://wiki.koeln.ccc.de/images/d/d5/Openchaos_qemudetect.pdf`. [Online].

Tseng, P.-H., Hsiu, P.-C., Pan, C.-C., and Kuo, T.-W. (2014). User-centric energy-efficient scheduling on multi-core mobile devices. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*, pages 85:1–85:6, New York, NY, USA. ACM.

UIAutomator (2019). Android ui automator. `https://developer.android.com/training/testing/ui-automator`.

Veeraraghavan, K., Lee, D., Wester, B., Ouyang, J., Chen, P. M., Flinn, J., and Narayanasamy, S. (2012). Doubleplay: Parallelizing sequential logging and replay. *ACM Transactions on Computing Systems*, 30(1):3:1–3:24.

Vidas, T. and Christin, N. (2014). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*, pages 447–458. ACM.

Vince (2014). Idc survey shows battery life is most important when buying smartphon. `http://blog.gsmarena.com/idc-surveys-50000-people-reasons-behind-buying-smartphone-battery-life-deemed-important/`.

Wong, M. Y. and Lie, D. (2018). Tackling runtime-based obfuscation in android with tiro. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 1247–1262, Berkeley, CA, USA. USENIX Association.

Xie, Q., Dousti, M. J., and Pedram, M. (2014). Therminator: A thermal simulator for smartphones producing accurate chip and skin temperature maps. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 117–122.

Xie, Q., Kim, J., Wang, Y., Shin, D., Chang, N., and Pedram, M. (2013). Dynamic thermal management in mobile devices considering the thermal coupling between battery and application processor. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 242–247.

Xu, E. (2017). Analyzing Xavier: An Information-Stealing Ad Library on Android. `https://blog.trendmicro.com/trendlabs-security-intelligence/analyzing-xavier-information-stealing-ad-library-android/`. [Online].

Xu, M., Bodik, R., and Hill, M. D. (2003). A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA, pages 122–135, New York, NY, USA. ACM.

Yan, K., Zhang, X., Tan, J., and Fu, X. (2016). Redefining qos and customizing the power management policy to satisfy individual mobile users. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12.

Yan, L. K. and Yin, H. (2012). Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium*, pages 569–584, Bellevue, WA. USENIX.

Yeo, I., Liu, C. C., and Kim, E. J. (2008). Predictive dynamic thermal management for multicore systems. In *2008 45th ACM/IEEE Design Automation Conference (DAC)*, pages 734–739.

Zhang, L., Bild, D. R., Dick, R. P., Mao, Z. M., and Dinda, P. (2013). Panappticon: Event-based tracing to measure mobile application and platform performance. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10.

Zhu, Y., Halpern, M., and Reddi, V. J. (2015a). Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149.

Zhu, Y., Halpern, M., and Reddi, V. J. (2015b). Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149.

Zhu, Y. and Reddi, V. J. (2013). High-performance and energy-efficient mobile web browsing on big/little systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24.

Zhu, Y. and Reddi, V. J. (2014). Webcore: Architectural support for mobile web browsing. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 541–552.

Zhu, Y., Samajdar, A., Mattina, M., and Whatmough, P. (2018). Euphrates: Algorithm-soc co-design for low-power mobile continuous vision. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA, pages 547–560, Piscataway, NJ, USA. IEEE Press.

CURRICULUM VITAE

[redacted]