Abstract of "Software Analysis and Development for Energy Efficiency in Mobile Devices", by Marcelo Teixeira Martins, Ph.D., Brown University, May 2017.

Smartphone and tablet users are sensitive to the battery drain exerted by mobile applications. In this dissertation, we focus on extending the battery lifetime of mobile devices. Using the perspective of a software engineer, we present a series of techniques that modify the behavior of mobile applications and increase their energy efficiency.

Our first contribution is Application Modes, a development aid that abstracts ancillary energy-management blocks (e.g., resource monitoring), thus allowing developers to focus on energy-related changes in software. Modes encapsulate code changes that modify the behavior of applications and their energy consumption. We show via examples that careful attention to application functionality substantially improves battery life.

An alternative to changing functionality at compile time is to do so at runtime. Our second contribution is Tamer, an execution controller that of monitors software events, rewires program binaries, and changes runtime behavior. Tamer interposes the execution of user-invisible energy-hungry software tasks, thus preventing their continuation. Developers can use Tamer as a tool to perform what-if analyses on the impact of battery life arising from potential code changes. Through a selective application of rate-limiting policies to demanding *apps* of the Android OS, we show that Tamer can effectively mitigate the excessive battery drain due to frequent wakeups of tasks running in the background.

A Tamer policy specifies which events to interpose and when actuation should occur. To write effective policies, developers must first understand which events are worth interposing. Our third contribution is Meerkat, an analysis tool that correlates software-execution logs with power traces. Meerkat leverages data-mining techniques to discover the interestingness of sequences of events and associates them with energy consumption. Meerkat helps developers uncover sequences of function calls that draw the most power, which will serve as input to effective policies. We demonstrate Meerkat's effectiveness in the domain of networked and I/O-intensive applications.

The insights delivered by our proposed techniques accelerate the investigation on the energy efficiency of mobile software. Given appropriate support, developers can better understand how to improve battery life and adopt combinations of software-adaptation strategies.

Software Analysis and Development for Energy Efficiency in Mobile Devices

by

Marcelo Teixeira Martins

B. S., Federal University of Minas Gerais, 2005

Sc. M., University of Tokyo, 2009

Sc. M., Brown University, 2011

A dissertation submitted in partial fulfillment of the

requirements for the Degree of Doctor of Philosophy

in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2017

This dissertation by Marcelo Teixeira Martins is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Rodrigo Fonseca, Advisor

Recommended to the Graduate Council

Date _____ _____

Steven P. Reiss, Reader

Date _____ _____

Sherief Reda, Reader

Date _____ _____

Justin Cappos, Reader
Computer Science and Engineering Department, New York University

Approved by the Graduate Council

Date _____ _____

Andrew Campbell, Dean of the Graduate School

Vita

2005	B.S. Computer Science, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, MG, Brazil
2005 – 2006	Research Assistant at SensorNet Lab, UFMG, Brazil
2006 – 2007	Student Researcher at Multimedia Communications Lab, University of Tokyo, Tokyo, Japan
2007 – 2009	Graduate Research Assistant, Department of Information and Communication Engineering, University of Tokyo, Japan Research Student Scholarship – Ministry of Education, Culture, Sports, Science and Technology of Japan
2009	M.Sc. Information Science and Technology, University of Tokyo, Japan
2009 – 2016	Graduate Research Assistant, Department of Computer Science, Brown University, USA
2010	Summer Research Intern, Intel Research Labs, Seattle, WA, USA
2011	M.Sc. Computer Science, Brown University, Providence, RI, USA
2013	Software Engineer Intern, Intel Corporation, Santa Clara, CA, USA
2016 – present	Systems Engineer, Apple Inc., Cupertino, CA, USA

Publications (Since 2009)

- Marcelo Martins, Justin Cappos, and Rodrigo Fonseca. *Selectively Taming Background Android Apps to Improve Battery Lifetime.* In Proceedings of the USENIX ATC, 2015.
- Qiang Li, Marcelo Martins, Rodrigo Fonseca, and Omprakash Gnawali. *On the Effectiveness of Energy Metering on Every Node.* In Proceedings of the IEEE DCoSS, 2013.
- Marcelo Martins and Rodrigo Fonseca. *Application Modes: A Narrow Interface for End-User Power Management in Mobile Devices.* In Proceedings of the ACM HotMobile, 2013.

- Hongyang Chen, Feifei Gao, Marcelo Martins, Pei Huang, and Junli Liang. *Accurate and Efficient Node Localization for Mobile Sensor Networks.* In ACM/Springer Journal on Mobile Network and Applications, 18(1), 2013.

- Marcelo Martins, Rodrigo Fonseca, Thomas Schmid, and Prabal Dutta. *Network-Wide Energy Profiling of CTP.* In Proceedings of the ACM SenSys, 2010.

Acknowledgements

First, I offer praise, worship, and thanksgiving to the all-holy Trinity of the Father, the Son, and the Holy Spirit for having given me the life, love, and grace to write and complete this doctorate. Second, I offer thanks and praise to the all-immaculate Mary for having aided me with her prayers and assistance. Third, I offer thanks to my namesakes: to St. Marcellus in whose name I was baptized in the Lord and to St. Thomas under whose patronage I received the sacrament of confirmation.

This doctoral dissertation is not a work solely of my own personal endeavor, for it could not have been written without the useful counsel, guidance, and involvement of a number of other people. I owe gratitude to my advisor, Prof. Rodrigo Fonseca, for letting me freely explore the paths of science under his guidance. Rodrigo is one of the few people I have met that is able to harmonize three important human virtues – intelligence, patience, and humility.

Thanks to Prof. Justin Cappos (New York University) for his sound words of wisdom and encouragement, especially when I was struggling with self-doubt. I am grateful to him for the long discussions that helped me sort out this work. I am also grateful to Prof. Steven Reiss and Prof. Sherief Reda for posing challenging questions that inspired me to continue exploring the subject of this work.

I am thankful to the secretaries and system administrators of the Department of Computer Science at Brown University for fostering a great work environment. In particular, I would like to thank Lauren Clarke, Jane Martin, Eugenia deGouveia, and Dawn Reed for their friendly support in taking care of all the important day-to-day matters during my stay in the department.

I appreciate the financial support from Brown University and Intel Corporation, who funded all my years of academic experience in the United States. I extend my gratitude to my colleagues at Apple Inc. for the words of encouragement during the last few months of this journey.

Thanks to all the lecturers and classmates who helped me cultivate a continuous interest in learning. A special appreciation goes to my comrades at the Systems Lab (Andrew Ferguson, Jeff Rasley, Jon Mace and Yu Da) as well as to some very estimated friends from the CS Department: Hammurabi Mendes, Irina Calciu, Alex Tarvo, Marek Vondrak, Yeounoh Chung, Erfan Zamanian, Cansu Aslantas, Sam Zhao, and Kaihan Dursun.

My sincere gratefulness goes to my sisters and other relatives who, even so far away, have always cheered for my success. Lastly, and most importantly, I want to thank my parents and my wife, Yoko, for their endless love, support, and life lessons. To them I dedicate this dissertation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Driven mostly by user needs, recent mobile devices have benefitted from a continuous expansion of their computing capabilities. Even so, a major hindrance to this expansion is the high operational demand and the very limited capacity of the battery included in these devices. Considering the interactive nature of mobile applications, we believe that the solution to this capacity problem should cover not only how mobile devices draw power but also what users can do given an expected battery lifetime. In this dissertation, we regard *battery lifetime* as the time a battery lasts before it runs empty.

Low-power research has focused on reducing the average power draw of computational tasks. Yet, it is equally if not more important to energy-optimize the user interaction with those tasks, i.e., to reduce the power draw driven by interaction and improve user productivity. After all, a short-lived battery leads to an unhappy user.

The central research question of this dissertation is how to increase the energy efficiency of software activities running on mobile devices so that battery lifetime can be extended and meet user expectations. The techniques we propose focus on the perspective of a mobile software engineer. This first chapter introduces the world of mobile devices and the power crisis that users of these devices face today.

1.1 The Rise of Modern Mobile Computing

Modern mobile computers encompass *smartphones*, tablets, and wearable devices including *smartwatches* and virtual-reality (VR) headsets[1]. Despite what their names might suggest, these devices are true computers, with an operating system (OS) that supports a wide range of applications and programming languages.

Yet, had they been mere small computers, they would hardly have presented much of an impact. Small portable computers were first promoted in the 90s and early 2000s in the form of "personal digital assistants" and "pocket PCs" from companies like Compaq (iPAQ [15]), Palm (PalmPilot [23]), and Apple (Newton [5]). None of these initial efforts lasted long. What changed is the fact that the most recent wave of mobile technology has brought to portable computers valuable features that surpass calling friends and taking notes:

Application model. Developers now have an economic incentive to develop and publish mobile applications, also known as *apps*. Building mobile apps is less expensive and labor-intensive than developing desktop software. Contrary to the recent past, developers now have free access to robust software development kits (SDKs) and extensive documentation. Furthermore, development frameworks like AppMachine [6], Apache Cordova [4], and AppMakr [11] help non-coders and beginner developers to quickly stitch software blocks together and deploy a complete app within hours. Developers also see a commercial incentive to release a mobile app: it can foster customer loyalty, rekindle interest in a company or brand, or introduce an additional sales channel.

Application market. The app-store (or marketplace) model, introduced by Apple in 2007 and successfully replicated since then, reduces the effort to discover and distribute new applications. Apps can be loaded locally with little to no external assistance or additional resources (e.g., another computer). Not only many applications are freely distributed in app stores, but the marketplace itself simplifies the purchase of paid apps, relieving publishers from much bureaucracy. The marketplace also ensures accountability and additional safety to customers by vetting apps free of malicious intent, holding developers responsible for misuse, and centralizing the distribution of updates containing new features and bug fixes. Finally, app stores provide a rating-and-review component that lets users voice their thoughts on the overall quality of an application.

[1] Although notebooks of various shapes and sizes (*ultrabooks* and *netbooks* come to mind) are also mobile devices, this dissertation solely focuses on the latest generation of handhelds.

Multi-touch user interface. Easy handling is key for a decent user experience. A touchscreen tends to be a very intuitive substitute for a keyboard and a mouse, resulting in easier hand-eye coordination [213]. Current touchscreen technology has vastly improved from the technology used in the 90s and 2000s.

With more reasons to succeed than before, modern mobile devices are now the least expensive computers with the lowest-cost applications, making them affordable to the majority of consumers. In its latest annual Mobility Report [12], Ericsson estimated a global number of 2.6 billion smartphone subscriptions for the year 2016, with an expected growth to 6.1 billion users by 2020. To many people, smartphones have become the standard platform to receive services and interact with acquaintances. Although they possess enough computing and communication properties to attend user needs, mobile devices are liable to a limited battery lifetime.

1.2 The Need for Power Management

User demands for more functionality and performance have led to an accelerated growth of sensing, computational, display, storage, and communication technologies for mobile devices. This growth has enabled a rich application environment that rivals the domain of desktop computers. Even so, battery capacity has not followed the same advancement pace.

Table 1.1 presents this technology gap in numbers. According to the International Technology Roadmap for Semiconductors (ITRS) [129], by the year 2029 the number of Application Processors[2] (AP) in a handheld will moderately increase by 6×. In the same time horizon, the number of cores in Graphics Processing Units (GPU) will increase by 50×, and the cellular data rate will grow at about 1.3× per year. Other technologies also expect significant performance improvements. Taking into account economies of scale, future mobile devices will require twice as much peak power to function.

Compared to these numbers, the battery-improvement rate is discouraging. Mobile devices have relied on lithium-ion batteries for the last two decades, with density improvements on the order of 5%-8% per year [81]. Despite the exponential growth, the density improvement rate is still smaller than the technology improvement

[2] An application processor (AP) is a system-on-chip (SoC) capable of running an OS and applications. The term is used to differ it from baseband processors of early cell phones, which were tasked with handling the communication over cellphone radios.

	Year	2015	2017	2019	2021	2023	2025	2027	2029
Input Metrics	# of AP cores	4	9	18	18	28	36	30	25
	Max CPU freq. (GHz)	2.7	2.9	3.2	3.4	3.7	4	4.3	4.7
	# of GPU cores	6	19	49	69	141	247	273	303
	# of Megapixels (display)	2.1	2.1	3.7	8.8	8.8	33.2	33.2	33.2
	Bandwidth (AP ↔ mem (Gb/s))	25.6	34.8	52.6	57.3	61.9	61.9	61.9	61.9
	# of sensors	14	16	20	20	21	21	22	22
	# of antennas	11	13	13	14	15	15	15	15
	Cellular data rate (Mbps)	12.5	12.5	21.63	40.75	40.75	40.75	40.75	40.75
	Wi-Fi data rate (Gbps)	0.867	0.867	0.867	7	7	28	28	28
Output Metrics	PCB area (cm^2)	62	69	76	84	93	103	103	103
	Peak power draw (W)	4.2	4.64	5.12	5.64	6.22	6.86	7.56	8.48

Table 1.1: Predicted mobile technology improvement for the next 14 years. Source: ITRS [129].

rate observed in Moore's law.

Although other battery technologies promise higher capacity per area, miniaturization challenges due to cost, physical and chemical constraints, and safety concerns make lithium-ion the most commercially viable solution to battery storage. Being so, it is important to use the available energy within a battery as efficiently as possible to meet user demands. Alternatives to extend battery life include reducing the power draw of hardware subsystems and designing more energy-efficient applications and operating systems.

To control the power draw of hardware subsystems, power management was introduced to personal computing in the early 90s [3] as a feature to shut down hardware, or to switch a subsystem to a low-power state, when inactive. In exchange for lower performance, hardware power management promises lower heat dissipation and lower cooling costs. Initially targeted at CPU cores, which were then responsible for the major part of a system's power draw, the initiative moved its attention to other subsystems. Still, as we will see in the next subsection, that is not sufficient to guarantee a long battery life in today's handhelds.

1.3 Is Energy Proportionality Enough?

Energy efficiency is broadly defined as the amount of performed computational work divided by the total energy used in the process. The concept of energy proportionality was christened in 2007 by Google's engineers Luiz André Barroso and Urs Hölzle [52, 53], who urged computer architects to design more energy-efficient

data-center hardware. Until that moment, hardware systems were mostly energy-efficient while inactive (sleeping or standing by) or while running at full speed, but not while lightly used. Achieving proportionality requires that most power-hungry components scale down their power draw according to demand.

This movement led to the research and development of various techniques that focused on the reduction of all energy-related costs of a data center, including capital costs, operating expenses, and environmental impact, as reported in the 2013-updated report by the same authors [51]. Still, as posed by Snowdon et al. [215], today's power management in mainstream OSes is conservative and simplistic. Standard policies are either the "race-to-halt" approach, which runs a workload to its completion at the maximum performance and then transitions to a low-power mode. Alternatively, there is an assumption that running at the lowest possible performance results in the highest energy savings. Instead, Snowdon et al. showed that either approach leads to sub-optimal results on real hardware.

Going beyond OS-level power management, other studies show that the energy efficiency of particular applications can be dramatically improved via changes in the program code, even in the presence of general-purpose power management [48,49,104]. Based on the argument that OS-level power management is application-agnostic, these studies exploit the workload variance of different applications to optimize hardware use. Additional research demonstrates that app developers may inadvertently write energy-inefficient code that causes rapid battery depletion without visible symptoms. Such suboptimal instances are known as *energy bugs* [178].

All of the preceding research examples show that it is crucial for application developers to be aware of the energy-consumption behavior of mobile devices. Without that ability, it is difficult to optimize program code for energy efficiency. This dissertation focuses on software-based solutions for analyzing and optimizing the energy consumption of mobile apps.

1.4 Approach and Contributions

"Soft" resources such as time (performance), space (memory or storage), and energy needed to accomplish a task are limited. Generally, a performance-oriented application consumes as much space and energy as needed

to shorten the running time and meet the program-output specification. When we switch our priority from performance to space or to energy, the roles of the other two resources are usually inverted.

Our approach to achieve software-based battery savings is to relax the requirement that mobile apps should pursue a single output specification. Instead, we allow a range of possible outcomes, where the range can be a few discrete states or a continuum. This range corresponds to different output quality levels, or utility levels.

To attenuate the power draw of mobile devices, we propose a cooperative system infrastructure that modifies the behavior of mobile applications so that they consume less energy depending on the context – application type, location, recharging opportunity, etc. These changes are inspired by the user's need to prioritize tasks. Although we explore the foundations of energy-aware software adaptation, this dissertation barely covers the decision-making process of when to enable those changes[3]. Instead, we focus on how developers can unleash more aggressive energy efficiency from applications.

Our proposed techniques encompass different approaches to modify the source code and runtime behavior of mobile apps, which we present as the main contributions of this dissertation:

Application Modes. Be an afterthought or a requirement in the software-development cycle, energy awareness necessitates additional care from programmers to guarantee that software consumes less energy. Application Modes is a development aid that abstracts ancillary, yet necessary, energy-management blocks (e.g., resource monitoring and actuation interface), thus allowing developers to focus on the energy-aware changes specific to their software. Application Modes adopt the idea of functionality bundles – encapsulated code-changes that modify the behavior of applications and, consequently, their energy profile – that a system can activate according to user preferences and the device's battery status. Chapter 3 elaborates on the motivation, design, implementation, and evaluation of Application Modes.

Tamer: An Energy-Controller for Background Tasks. An alternative to changing software functionality at compile time is to do so at runtime. Tamer is an execution controller capable of monitoring events, rewiring program binaries, and changing runtime behavior without need of source-code correction, recompilation,

[3]Such topic would require a dissertation by itself.

or software re-installation. TAMER works by interposing the execution of software events and preventing their continuation. In Chapter 4, we discuss the internals of TAMER and demonstrate how to link an event-throttling controller with effective policies to reduce the excessive battery drain due to frequent wakeups of software tasks running in the background.

Meerkat: A Policy-Discovery Tool for Energy Controllers. While Chapter 4 focuses on the controller part of TAMER, Chapter 5 concentrates on the policy engine that drives TAMER's actuation. A TAMER policy specifies which software events to intercept and when actuation should occur. Before developers can write energy-effective policies, they need to understand what events are worth intercepting.Chapter 5 presents Meerkat, an analysis tool that correlates software-execution logs with power traces. To associate energy consumption with event bundles, Meerkat leverages sequential-pattern mining [37] to to discover the interestingness (e.g., frequency, usefulness, novelty, etc.) of sequential patterns. A sequence energy-rule tells the occurrence likeness of a subset of sequential events as well as its energy impact. Developers can use Meerkat to discover the sequences of events that consumes the most energy and use them as input to write effective TAMER policies.

Case Studies: Specialized Application of TAMER and Meerkat. Meerkat's effectiveness depends on the types of events being monitored. TAMER's original design focuses on event dispatching within the Android OS. Because of the generality of those events, sometimes it is difficult to discern the utility of valuable event-sequences. In Chapter 6, we address this challenge by proposing specialized TAMER payloads that monitor and intercept additional subsets of events. These events subsets not only include functionality within the Android OS but also dive into popular third-party libraries, thereby facilitating the identification of unique energy-hungry event sequences and their utility by Meerkat. We describe the design and implementation of two TAMER-based controllers – NetDiet and IODiet– that specialize in tasks involving networking and storage events. Finally, we evaluate their effectiveness in curbing excessive battery drain.

Before we explore each contribution, the reader might want to familiarize herself with the energy-management process in mobile systems as well as the major influences in the field that led to the development of our research. We dedicate Chapter 2 to this task.

Chapter 2

Background

This chapter presents the background to power management in modern mobile devices. We have limited ourselves to the work that is essential to understand the topic of this dissertation. As this dissertation focuses on *software* solutions to improve energy efficiency, techniques involving low-power hardware design, such as transistor sizing [61, 93], transistor reordering [66, 185], clock gating [231], and power gating [123, 226], are out of scope. We acknowledge the contributions to the field during the early 90s and 2000s and focus on the latest relevant works targeting modern mobile devices.

2.1 Overview

Energy efficiency in mobile software can be achieved at different levels. The main challenge question is: *Who should be responsible for energy management? Applications or the OS?* We believe that, in many cases, both actors should play an active role. We structure our study on energy management and optimization using a taxonomy of different types of solutions.

- *Energy-aware operating systems.* At the OS level, the main idea behind reducing energy consumption is to unify resource management with energy management and to promote a collaboration between applications and the OS. Having a good understanding of how resources are demanded by users and

applications is crucial to achieve proper energy efficiency. Section 2.2 focuses on energy-aware mobile OSes and resource management.

- *Energy profiling.* Energy awareness calls for understanding how hardware components consume energy. Without knowledge of how subsystems and applications drain the battery, users, developers, and the OS cannot decide on how to improve battery life. Section 2.3 elaborates on the research done in this field.

- *Software analysis and optimization.* Understanding and detecting energy bugs and hotspots in applications for smartphones and tablets is paramount to developers. To reduce energy consumption, developers can resort to context information from subsystems and applications to extract use patterns and optimize resource management. Section 2.4 discusses techniques both related to software analysis and optimization.

- *User interaction and its impact on battery life.* Battery lifetime is a usability challenge. Improvements in energy efficiency should also take into account how users interact with batteries and other device resources. An energy-aware system should know when, where, and how users drain batteries, and if there are recharging opportunities. Section 2.5 dives into some of the recent contributions to this area.

- *Computation offloading.* Modern mobile OSes often rely on online services running remotely. Remote execution allows migrating computation from battery-powered mobile devices to wall-powered, highly performant servers hosted remotely. Factors like latency and data-metered networking act as counterparts to the benefits of offloading. Section 2.6 covers some of the relevant works in mobile-workload offloading from an energy perspective. Interested readers should check the survey by Fernando et al. [101] for a more thorough overview of this area.

2.2 Energy-Aware Operating Systems

Power management in current mainstream operating systems is simplistic. An example of a standard policy is the "race-to-halt" approach, which runs a workload to completion at maximum performance and then

transitions the hardware back to low power. Academic prototypes of more advanced systems have shown the benefits of more aggressive energy-oriented policies.

Going beyond the idea of energy proportionality, the first academic, energy-aware OSes for personal devices appeared in the late 90s and early 2000s. Odyssey [172] and ECOSystem [243] follow a hybrid approach in which both applications and the OS collaborate to reduce the system's power draw. In 1999, Carla Ellis proposed energy to be considered a first-class resource, augmenting the traditional OS perspective of focusing solely on performance [95]. With the rise of smartphones and tablets by the end of 2000s, the topic regained attention due to the alarming rate at which power-hungry software have been reducing the battery life of handsets.

In Odyssey, applications adapt the Quality of Service (QoS) delivered to users based on the available battery state and resources. Odyssey monitors resource demands and convey to applications changes based on the concept of *fidelity* – the quality of the software output. Odyssey estimates the energy and resource demands of applications by observing their execution history and battery feedback. Odyssey leverages online profiling via PowerScope [105] to estimate the system's energy consumption (cf. §2.3).

ECOSystem takes a different approach: it allocates energy to competing tasks using the *currentcy* unit abstraction [244] and schedules their execution taking into account fairness and the tradeoff between performance and energy consumption. To do so, ECOSystem uses *resource containers* [50] to isolate and track the resource demands of applications.

Another example of OS-application cooperative approach is GRACE-OS, by Yuan and Nahrstedt [237, 238]. The authors used the probability distribution of CPU utilization by applications to design a specialized task scheduler that is able to set the appropriate CPU-core voltage. According to its authors, GRACE-OS can achieve energy savings of up to 37.2% on mobile multimedia workloads.

Koala [215] is a platform for OS-level power management that uses a pre-characterized model to predict the CPU performance and energy consumption of software blocks. At each context switch, Koala pairs the application's observed behavior with the system's pre-defined policy to determine the most appropriate power settings for the CPU and memory frequencies. Koala achieves near-optimal energy savings for single-threaded, CPU- and memory-bound tasks.

Cinder [198] is a mobile OS, built atop the HiStar exokernel [240], that employs device-level resource accounting and power modeling. Cinder aims to achieve effective energy allocation to applications using three principles: isolation, subdivision, and delegation. Similarly to ECOSystem, Cinder allocates energy to applications using a discharge-rate metric. Cinder applications can also collaborate and share their energy reserve.

Despite the various academic prototypes of OSes containing radical architectural changes, many of these contributions have never reached consumer devices for reasons ranging from implementation cost to lack of bounded performance/power guarantees. Nevertheless, less radical OS power-saving mechanisms have found their way into commercial mobile devices. Because these mechanisms are complimentary to our work, we briefly mention them here. These include smart task scheduling [110, 190, 228], voltage scaling [156, 183, 184], frequency scaling [114], and dynamic voltage and frequency scaling (DVFS) for CPU cores [77, 130] and GPU cores [35, 143]. Scheduling for heterogeneous CPU multi-cores [8, 82] and transferring tasks between CPU and GPU cores [56, 157] are other examples of techniques that rely on the cooperation between the OS and the underlying hardware to achieve performance gains or energy savings.

2.3 Energy Profiling

Efficient power management in mobile platforms is a complex and challenging research problem due to the multitude of hardware configurations and power states and their interdependence caused by multiple applications. A core requirement of effective and efficient energy management is a good understanding of *where* and *how* the energy is used – how much of the system's energy is consumed by which parts of the system and under what circumstances. We can obtain such knowledge from direct measurements or from modeling.

2.3.1 Measuring Power

Developers can resort to external instruments to directly measure power. Oscilloscopes, multimeters, and Data ACquisition (DAC) hardware from different vendors can sample the voltage or current flow of the power supply. While these methods vary in terms of accuracy and resolution, most of them require soldering onto or

cutting into the battery connection of a mobile device. The Monsoon Power Monitor [169] is a self-powered portable commercial solution that connects to the device's battery to measure the current flow without need of soldering. BattOr [206] consists of a small non-invasive circuit board that, while interposing between a device and its battery, collects accurate power samples at a high frequency (10KHz). Both Monsoon and BattOr assume that the battery is reachable and detachable. This assumption contradicts the current seamlessness-focused design trend in premium smartphones, which precludes making their batteries user-accessible [16]. Yet, because external power monitors are the gold standard for power analysis, we base our findings in this dissertation on direct measurements.

A second approach to measure power is to query the battery state. Mobile devices include special-purpose electronics embedded in the mobile device – the fuel gauge. Software can query self-metered battery data using a special API. Depending on the fuel-gauge type and the device model, different battery data can be queried. For instance, the Google Nexus 6 smartphone uses a Coulomb-counter fuel gauge that periodically informs the battery's current draw, voltage, and temperature [17].

A series of studies use direct measurements to profile mobile hardware and software. One of the most detailed and general analyses of energy consumption in modern mobile devices is the report by Carroll and Heiser [67]. The authors presented a detailed breakdown of the power draw of smartphone subsystems using an external high-resolution power meter. After detailing the energy impact of the CPU, memory, touchscreen, GPU, audio, storage, and network interface, Carroll and Heiser concluded that the most energy-hungry components in a smartphone are the display and GSM (radio) modules. Although their study dates back to 2010, their findings are still relevant.

Using a multimeter, Thiagarajan et al. [220] characterized the energy needed to render web pages on a mobile browser. The authors found that downloading and parsing cascade style sheets (CSS) and JavaScript consume a significant fraction of the total energy needed to render a web page. The authors also proposed recommendations on how to design web pages in order to minimize the energy of their rendering.

NEAT [63] is a toolkit containing a coin-sized power-measurement board, which fits inside a typical smartphone, and analysis software that automatically fuses the event logs taken from the device under test with

measured power samples. NEAT was created after its authors demonstrated that power-estimation models can raise inaccuracy errors by 20%. McCullough et al. [162] also demonstrated the limitation of several advanced power-modeling techniques posed by hardware complexity and variability, thus motivating the need for direct measurements.

2.3.2 Modeling Power

Because direct measurements do not scale well, power modeling is a popular technique used to associate power draw with running tasks. A *power model* is a mathematical representation that quantifies the impact factors of energy consumption due to running software, such as the utilization level of a hardware subsystem. A power model can characterize a single subsystem, a combination of them, or even a whole device (system-level model). Approaches to modeling power draw can be divided into two categories.

Utilization-based models. Utilization-based models account for the energy consumption of a subsystem by correlating the power draw of that subsystem with its measured usage rate. Given a running application, a valid power model includes variables that reflect all of the different components that are active.

To obtain a utilization power model, a training phase is required. A modeling tool collects the utilization of individual hardware components, typically via an OS-hardware interface, from running sample applications and measuring the corresponding power draw of those components. Next, the modeller uses machine learning to derive a model that correlates the utilization samples with the measured power draw.

Early models focused on estimating the power draw of individual subsystems. For instance, Bellosa [55] associated data from hardware *performance counters* – instructions per cycle (IPC), fetch counters, miss/hit counters, stalls – with CPU energy consumption to obtain an estimate of the energy use at each CPU-core frequency. The intuition is that the amount of power required to execute instructions at a given frequency is proportional to the amount of activity inside the microprocessor. Bircher and John [58] explored the use of counters to estimate the power draw of other subsystems.

Flinn and Satyanarayanan's PowerScope [105] maps energy consumption to program structure in a similar

fashion to CPU profilers such as the `prof` command in UNIX machines – using statistical samples of execution traces aligned with metered power draw.

More recent utilization models focus on attributing system and subsystem energy consumption to code blocks. pTop [89] is a process-level power profiler for devices running the Linux OS. pTop uses process-context and hardware-utilization data obtained from the OS kernel, along with power-specification data from hardware vendors, to estimate the energy consumption of running applications. A drawback of pTop is the lack of apparent support for workload variation.

PowerTutor [241] is a profiler that assigns power draw to mobile applications by monitoring the activity state of subsystems. PowerTutor relies on PowerBooter, an automated power-model builder that uses built-in battery sensors and knowledge of battery-discharge, to estimate the energy consumption of individual subsystems.

Quanto [106] uses the iCount energy meter [94] to record system energy consumption and relies on linear regression to generate power models that estimate the energy intake of the hardware components of an embedded device. Quanto tracks the intervals in which components are operating on behalf of programmer-labeled activities and associates the estimated energy with those activities.

Because power models work as an approximation of the real behavior of hardware components, they are prone to inaccuracy. External factors can affect the battery-drain rate, such as device temperature, battery age, and faulty hardware, thereby necessitating model recalibration. Sesame [91] and V-Edge [232] sidestep the need for external metering to recalibrate a power model. By collecting fine-grained voltage data from the built-in battery interface along with select microbenchmarks, both approaches can self-construct new models to reflect the actual power draw of hardware under different conditions.

Event-based models. Several components in mobile devices, such as GPS, storage, and camera, act like black boxes and do not expose quantitative utilization metrics, making it hard to obtain accurate models using the aforementioned approaches.

Balasubramanian et al. [49] presented a measurement study of the energy consumption characteristic of three widespread network technologies: 3G, GSM, and WiFi. The authors found that the 3G and GSM radios incur

a high *tail-energy* overhead due to the lingering of high-power states long after completing a packet transfer. Tail energy cannot be easily attributed when using a utilization-based model.

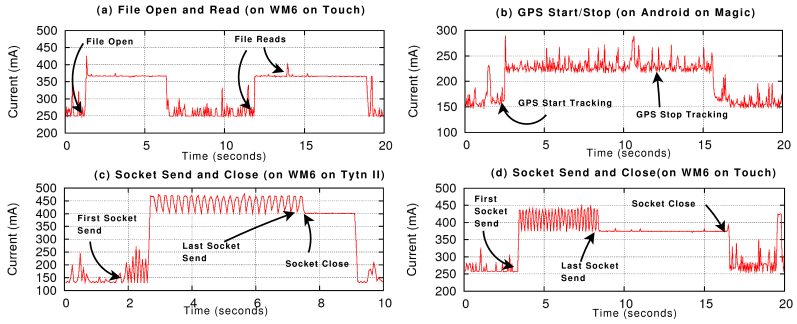


Figure 2.1: Tail-energy overhead of asynchronous subsystems. Source: [180].

Pathak et al. [180] observed that the same tail-state effect prevails in other subsystems, such as GPS and storage (Figure 2.1). The authors followed with a new power-modeling approach based on the tracing of application-initiated system calls that lead to hardware activation, Their approach gracefully captures both the utilization-based and the non-utilization-based power behavior of I/O components. The new models rely on finite state machines that indicate the current state of a hardware subsystem, along with transitions to other states due to expired timers or other external conditions. The authors demonstrated that event-based models can achieve high accuracy in fine-grained power estimation.

Using this approach as a basis, Pathak et al. introduced *eprof* [179], the first fine-grained energy profiler for smartphone apps. *eprof*'s main contribution is the capture of energy dissipated by asynchronous tasks, a feature not available in previous profilers.

Similarly, Qian et al. [188] proposed a methodology that accurately infers the Radio Resource Control (RRC) states of a 3G cellular modem and the associated power draw from packet traces collected on a handset.

Finally, Carat [173,174] and eStar [71] are two different approaches to energy apportioning using the same principle: use battery data crowdsourced from thousands of smartphone and tablet users to estimate the average impact of individual application on battery drain.

Power modeling is a vast area of research that has seen many contributions over the years. We refer interested readers to Hoque et al.'s survey [121] for more information on the area.

2.4 Software Diagnosis and Optimizations for Energy Savings

It is difficult for software developers to measure the energy consumed by their applications and to explore how this energy consumption might change with conditions that vary outside the developer's control. In parallel, mobile software present patterns that can be exploited for the sake of energy savings. In the following, we present some of the energy-focused techniques that explore the optimization of resource management based on context changes.

Pathak et al. [181] presented a comprehensive study of no-sleep energy bugs in Android-OS applications. The authors resorted to dataflow analysis on decompiled Java bytecode to uncover and debug a series of bugs that keep subsystems awake due to programming mistakes and race conditions. Experiments show that their analysis tool can detect previously reported and unreported no-sleep bugs with a low rate of false positives.

WattsOn [166] is a plug-in to the Visual Studio IDE [32] that allows developers to estimate the energy consumed by code blocks. WattsOn relies on event-based power modeling to estimate energy consumption and on software emulation to scale *what-if* analyses on code changes. eLens [117] uses program analysis to determine code paths traversed during execution, and per-instruction energy modeling to obtain fine-grained estimates of application energy. Li et al. [147] also used program analysis to estimate the energy consumed per line of source code of various mobile applications. Both eLens and Li's work claim accuracy errors within 10% of ground-truth measurements.

Simulating typical use scenarios from 55 mobile apps, Linares-Vásquez et al. [151] presented a quantitative and qualitative empirical investigation on energy-greedy APIs and software patterns in the Android development framework.

IMP [119] is a network data-prefetching library to which mobile applications can link. IMP employs a goal-directed adaptation mechanism that decides when to prefetch data on behalf of mobile apps and tries to

minimize application response while meeting budgets for battery lifetime and cellular-data use. IMP opportunistically uses available networks while ensuring that prefetching does not degrade the network performance of foreground activities. RadioJockey [44], TOP [187], and Bartendr [205] are also energy-aware packet schedulers that leverage patterns in data streams to reduce the expended energy of radio-transmission tail states.

A-Loc [150] is an energy-aware middleware for location-based apps of the Android OS. A-Loc incorporates probabilistic models of user location and sensor errors to pick the most energy-efficient sensor that meets the position-accuracy requirements of applications. EnTracked [137] estimates and predicts the system's state and user mobility to better schedule the GPS sensor to obtain position updates. EnTracked promises potential energy savings between 40% and 50% compared to periodic GPS sampling, with a maximum, yet unlikely, error of 200m.

Ghosts in the Machine [38] suggests giving the OS more visibility on the power state of I/O devices. Developers adapt applications to hint about their execution to the device's power manager, Affected subsystems adapt their power-state schedule, resulting in performance and energy-conservation improvements. Users define QoS using a unit-less knob that prioritizes performance or energy savings. We present a similar concept of cooperative power management in Chapter 3.

In the realm of programming languages and frameworks, there is a growing interest in facilitating the developer's life by introducing new language features that provide cues to the compiler and the OS about opportunities for saving energy [78,199]. Recent techniques, like task discarding [194] and loop perforation [212], comprise a larger set of mechanisms that can be used in programming-language semantics and execution environments to reduce the demand for resources. Eon [217] and Energy Levels [140] offer programming abstractions and runtime environments that predict resource use in wireless sensor networks and automatically meet lifetime goals by deactivating or adapting parts of an application. In the domain of approximate programming, Green [47] let programmers define statistical QoS guarantees by approximating expensive functions and loops.

2.5 Users and Batteries

An important way to characterize handheld use is to quantify the value that users place on their device battery. A large-scale longitudinal study by Ferreira et al. [102] explored the charging habits of more than 4,000 smartphone users and found that users charge their devices frequently throughout the day. The study showed that users perceive battery drain as an obstacle to obtain value from applications. The following studies explore the interaction between users and batteries in different ways.

CABMAN [191] is a battery-management architecture for mobile devices that uses context information (location and time) to predict the next recharging opportunity and to inform users about the remaining running time of critical (e.g., telephony) and non-critical applications. As a result, users have a better sense of which tasks to prioritize before their battery runs out.

Shye et al. [210] developed a logger application for Android mobile devices and collected 250 days of activity data from 20 users. Using a regression-based power model, the authors discovered that energy consumption varies widely. Their study concludes by stating that the CPU and screen are the two largest battery-drain sources and that, on average, 49.3% of the total system power is spent while idling. Other works [72, 233] confirm this discrepancy between active and inactive periods. Chapter 4 focuses on optimizing the energy consumption while the system is supposedly idling.

Falaki et al. [97] conducted a comprehensive study of smartphone use based on detailed app-activity traces from 255 users of Android and Windows Mobile devices. The authors characterized intentional user activities and how these activities impact network and energy use. Coinciding with Shye's findings, Falaki et al. uncovered an immense diversity among user activities and suggested that mechanisms to improve user experience and energy consumption should learn from and adapt to user behavior.

Truong et al. [221] discussed the coarse granularity and inaccuracy of today's battery interfaces – overall percentage or amount of time remaining – and how users find it difficult to trust those interfaces. From the analysis of responses to a target survey of 104 users, the authors designed and prototyped a task-centered battery interface that shows more accurate information about how long individual and combinations of tasks

can run before battery depletion. A pilot study with eight users showed that the proposed interface can help users better understand battery use and be more effective in selecting which tasks to run. Ferreira et al. used the same motivation to create IBI [103], an interactive interface to help users extend battery life.

These studies on the interaction between users and batteries unveil a common trend: the need for a system-centric and user-centric resource management system that leverages contextual and fine-grained information to make decisions.

2.6 Task Offloading

Computation on mobile devices usually requires compromises. Although mobile hardware continues to evolve, it is still not in the same performance category as desktop and server hardware. Improving size, weight, and battery life are higher priorities than computational power. Yet at the heart of many popular smartphone apps and tasks, high performance is desirable. These include speech recognition, natural language processing, computer vision and graphics, machine learning, and augmented reality, to name a few.

Remote computing comes as a natural solution to the resource poverty of mobile devices. Instead of executing all tasks locally, some are offloaded to execute on a remote server or another portable device over a network connection. Many of the official Google and Apple mobile apps for the Android and iPhone platforms follow this execution model using the power of cloud computing.

Most of the attempts of remote execution take one of the following approaches: (1) Developers partition a program, specifying what state needs to be transferred, and how to adapt the partition scheme to the changing network conditions; (2) Developers use full process [175] or full virtual-machine (VM) migration [76, 200] to reduce the development burden.

While remote execution helps mobile devices save energy, systems leveraging this technique should be aware of its shortcomings. Network delays can hurt usability by degrading the crispness of system response. Loosely coupled tasks, such as Web browsing, might continue to be usable, but deeply immersive tasks, such as augmented reality, can become sluggish to the point of distraction. Lagar-Cavilla et al. demonstrated that

latency can quickly impact interactive response in spite of adequate bandwidth. Even a modest latency of 33ms causes the frame rate of a remote visualization app to drop considerably from that experienced locally [141]. In addition, battery-constrained devices should not offload tasks if the energy required to synchronize application state is higher than executing tasks locally.

CloneCloud [76] offloads parts of the application workload to the cloud by using VM migration. CloneCloud rewrites mobile apps before migrating threads, chosen automatically, to a cloned device in the cloud. When tested on select applications (virus scanning, image search, and behavior profiling), CloneCloud showed speedups between 12× and 21×, and energy savings between 4× and 14×, while requiring an extra bandwidth of 0.4Mbps and latency of 680ms.

To avoid full VM migration, MAUI [84] uses a fine-grained approach to offload parts of an application to the cloud. MAIU decides at runtime what methods to remotely execute, while considering energy savings and connectivity constraints. MAIU relies on code portability to quickly deploy multiple versions of the same application, on managed code to abstract architecture differences – ARM (mobile) vs. x86 (server) –, and on code annotations to hint on context switching. Similarly to CloneCloud, MAUI automatically identifies costs through static and dynamic code analysis.

Chroma [48] semi-automatically partitions an existing application into remote and local tasks by taking advantage of application-specific knowledge. Chroma takes as input all the meaningful partitioning strategies of an application, specified by developers in a declarative form (*tactics*), and selects an appropriate partitioning using online prediction of resource demands. The tactics abstraction captures software knowledge relevant to remote execution with minimal exposure of the implementation details. This allows the use of computationally intensive applications on handhelds even in environments with ever-changing resource availability.

Rather than connecting to a distant cloud, Satyanarayan et al. [200] proposed the use of cloudlets – several smaller and less expensive computers installed in common areas (e.g., coffee shops, schools, etc.) – to serve as proxies to remote cloud servers. Mobile devices connect to a physically close, resource-rich cloudlet using a low-latency, one-hop, high-bandwidth wireless connection, thereby guaranteeing real-time interactive response. Like CloneCloud, cloudlets rely on VM migration. Still, cloudlets avoid the high price of networking

via *dynamic VM synthesis* – a mobile device delivers only a small VM overlay to the cloudlet that already possesses the base VM from which the overlay was derived. Several computing-intensive applications have been successfully ported to the cloudlet infrastructure [116, 202, 214].

Collaborative computing is another approach used to facilitate task offloading. IDEA [69] is a sensor-network service for effective network-wide energy decision making. IDEA distributes information from each node on battery level, charging rate and execution state, and helps networked nodes evaluate each remote-execution option using an energy-objective function tailored to meet specific application goals. ErdOS [224] is an extension to the Android OS that predicts the resource demands of mobile apps based on user's habits and preferences, and leverages access to computing resources of nearby devices via wireless communication. ErdOS uses social connections between users to enable and disable access to remote resources.

A common theme in all of the aforementioned solutions is the use of application knowledge in different forms to facilitate and optimize task-offloading, thereby saving energy. As we will demonstrate in Chapters 3, 4, 5, and 6, we use the same software-knowledge intuition and develop different strategies to optimize the battery drain of mobile applications.

2.7 Summary

Since the mid-90s, researchers have been promoting energy as a primary system resource for mobile devices. In this chapter, we presented a snapshot of the research conducted in the field of resource management for energy efficiency in mobile systems. We separated the relevant work in five categories: energy-aware operating systems, power measurements and models, software analysis and optimizations, user interaction with mobile resources, and opportunities for task migration and remote processing.

Smart power management often requires certain actions to be deferred, avoided, or slowed down to prolong battery life. These opportunities are best explored if we consider application-context knowledge and the participation of multiple agents (users, OS, and developers) in defining how the battery should be used. In the following chapters, we present our approach to this multi-faceted challenge.

Chapter 3

Application Modes

Exposing Application-Specific Internals for Energy Control

3.1 Introduction

Although necessary, optimizing energy consumption during system idling is of little use if subsystems are often demanded, especially if we consider the increase of background services and complex applications that keep a system awake. Being so, it is necessary to reduce resource demand to achieve energy efficiency. Fundamentally, under some circumstances, there has to be a prioritization of functionality, as the energy density in handheld batteries is not sufficient to perform all possible system functions continuously.

Many past projects recognize the different roles of the OS, applications, and end-users in power management (PM). Odyssey, a project by Flinn and Satyanarayanan from the late 90s, was the first to simultaneously involve the OS, the applications, and the user in power management [104]. In Odyssey, applications are guided by the OS to dynamically change their behavior, limit their energy consumption, and achieve user-specified battery lifetime. The adaptation involves a tradeoff between energy use and application data-quality, to which the authors refer as *fidelity*. Fidelity is application-specific and opaque to the OS. The role of the OS is to direct

the adaptation based on supply and demand of energy by the device and its relation to the expected battery duration. When the OS detects that the lifetime goal cannot be achieved, the system issues upcalls to applications in order to reduce their fidelity. The user participates by inputting two pieces of information: the desired lifetime and a prioritization of applications to sort the adaptation options. Finally, application developers are responsible for implementing different fidelity levels.

Many factors make it opportune to revisit Flinn and Satyanarayanan's original work. Due to a combination of more complex applications, a plethora of devices to choose from, and a diverse user base, in some cases there is no single fidelity metric that is common to all users in all contexts. As a result, automated approaches can only suboptimally adapt some applications. Furthermore, given advances in hardware and lower-level software (e.g., ACPI), devices are much more efficient when idle, making high-level approaches that reduce active demand much more effective now than a decade ago.

A fundamental assumption in previous works [48,104] is the existence of a well-defined tradeoff between fidelity (or QoS) and energy use. Consequently, an application developer knows the app configurations that lie in the *Pareto frontier* of this tradeoff, enabling an automated algorithm to decide the optimal state based on the available battery life.

Even though this assumption holds true for many applications, there are counterexamples. As we show in §3.2, two users with different priorities could aim for different tradeoffs between energy use and *utility* from an application. The key observation is that in these (and other) cases, automated adaptation may fail and the runtime system must ultimately elicit preferences from the user. The main challenge is how to involve the user at the right time and *at the right level*. Participating users should only worry about tangible aspects of the device operation, such as lifetime and functionality, and not be concerned with how they are implemented.

We present *Application Modes*, an interface between applications and the OS that eases the separation of roles for effective power management. Rather than exposing a metric, application developers declare to the OS one or more *modes* composed of reductions of functionality with presumed power savings. For instance, a shooting game could provide two modes: a high-quality mode that strives to keep the user experience smooth (i.e., keep the frame rate close to 60 frames per second (FPS)), and a power-saving mode that reduces the

Profile Title	Encoding Settings	Network Output (MB)	Live Streaming?	Software Used
HD Streaming	720p video (H.264), hi-def audio (AAC)	158.60	Via RTSP	LiveStream
HD Recording	720p video, hi-def audio	290.1	Upload on recharge	SpyCam
SD Streaming	480p video (H.263), med-def audio (AAC)	47.16	Via RTSP	LiveStream
SD Recording	480p video, med-def audio	183.4	Upload on recharge	SpyCam
Audio Recording	Audio only (AMR-NB), screen off	0.58	Upload on recharge	Sound Recorder

Table 3.1: Functionality alternatives for the media-streaming app. Upper profiles yield higher-quality videos in exchange for larger network output.

frame-rate requirement to 30FPS. Modes carry a human-readable description of the assigned functionality and the promise of switching when requested by the OS. We assume that the OS can predict how long a device's battery will last at each mode, and then request a mode change whenever appropriate. However, recognizing that different modes may have different utilities for different users, we believe that the decision of when to switch modes should involve the user whenever necessary. This involvement takes form in combining the description of each adaptation with the prediction of lifetime changes by the OS.

3.2 Motivation

In this section we use power measurements on a common handheld workflow scenario – media streaming on a smartphone – to illustrate two points. First, we confirm and extend the earlier findings by Flinn and Satyanarayanan [104], which demonstrated how changes in application behavior can substantially affect energy consumption. Second, we show that different users can have very different Pareto frontiers in the utility-energy tradeoff, making globally automated decisions ineffective in maximizing utility.

We measure the power draw of running a streaming application using very different *modes*, or bundle of settings. We performed our measurements on a Samsung Galaxy Nexus smartphone running a customized version of the Android Jelly Bean OS (4.3) while connected to a Monsoon power monitor. To discriminate the power draw of the inspected application, we first measured the phone's power draw in the *idle* state, i.e., not running any applications apart from the base system, and established two baselines: one with the screen on and another with the screen off. We made sure that the screen brightness was constant in all runs.

The media-streaming app records and transmits audio and video over the Internet. This scenario is a

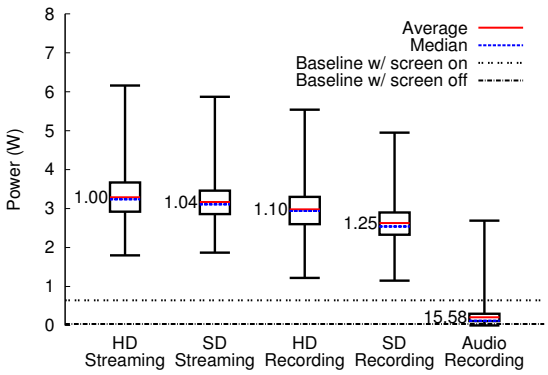


Figure 3.1: Power draw distribution of different profiles for the media-streaming app (cf. Table 3.1). Box limits represent the first and third quartiles and include the median and average power draw. Whiskers indicate the minimum and maximum power draw. Numbers to the left of each box show the improvement in battery drain relative to "HD Streaming." Greater energy savings can be achieved by reducing the media output quality and immediacy.

common user case and presents interesting tradeoffs between utility and energy savings, as different hardware subsystems are activated. For instance, a user documenting an important event may favor immediate uploading of a low-quality video, whereas another user may prefer a high-quality capture to be uploaded only when connected to a power outlet. Selecting the right application parameters for each of these cases can, however, be a daunting task to the average user.

To illustrate the tradeoffs we consider the power draw of recording and uploading a five-minute video feed using three similar applications. Using the hardware setup described above, we simulated different profiles using the following applications: SECuRET LiveStream [28], SpyCam [29], and MIUI Sound Recorder [18].

In all cases we modified the video- and audio-encoding parameters as well as considered different moments to upload the recorded media. We derived five bundles of settings, listed in Table 3.1. Once again, these settings are *not* transparent to the user, and offer specific tradeoffs between quality and timeliness of the uploaded media.

Figure 3.1 shows the power draw of each configuration set. "Audio Recording" draws on average over 15× less power than "HD Streaming". "Audio Recording" also outputs the least amount of bytes, does not use the screen, camera nor video-encoding hardware, and defers network I/O to when the device recharges its battery.

Figure 3.2 shows the same modes, with a numerical utility for hypothetical users (which could be even

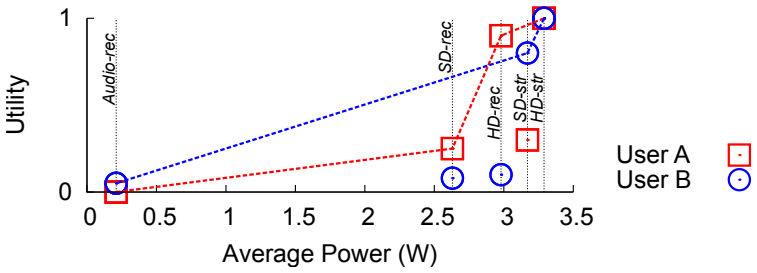


Figure 3.2: Utilities of different streaming modes for two users of the streaming application. 'A' prefers a high-quality capture, whether streamed or recorded, whereas 'B' values immediacy over quality. The Pareto frontiers (dashed lines) are different, and no single policy can choose between "SD Streaming" and "HD Recording" for all users.

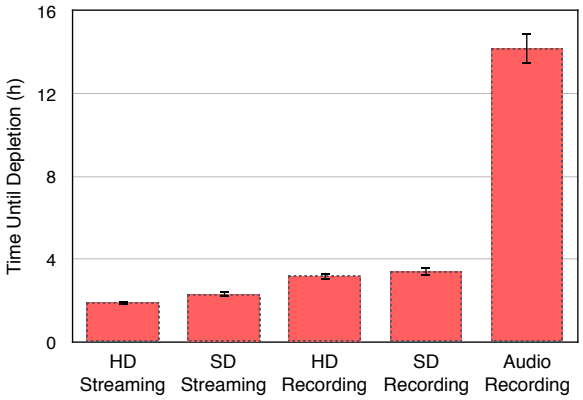


Figure 3.3: Battery lifetime for different streaming settings.

the same user in different contexts). User 'A' is interested in obtaining high-quality video, whereas user 'B' values immediacy. The graph shows that the Pareto utility frontier for each user is different, and that there is no consistent ordering of modes, particularly between "HD Recording" and "SD Streaming". This example highlights that neither the OS nor the application can know a priori the utility of modes for each user. In this case, and in general when there are multiple dimensions that users value differently, the automatic selection of a mode breaks down.

Finally, picking the most appropriate bundle of settings depends not only on the assessment of feature tradeoffs, but also on how much battery life users plan to save, if that is their priority. Figure 3.3 shows the complete battery lifetime from running the streaming workload at each proposed configuration. Not surprisingly, a device that uses less resources draws less power and its battery lasts longer.

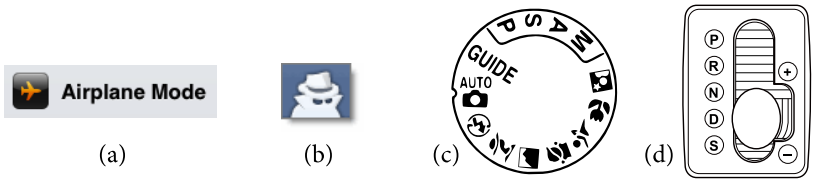
3.3 Separating Roles in Power Management

In this section we argue why users, applications, and the OS should all limit the active demand of resources by mobile devices to achieve maximum *value* out of a limited energy budget.

1. The OS cannot always know the resource priorities of all applications. If an application consumes too much energy, the OS could limit its access to system resources, such as CPU time, bandwidth, and accurate location fixes. Robust applications should bear utility reductions and adapt themselves. However, arbitrary utility reduction can be frustrating to the user, *as it is hard to estimate her sense of value for functionality*. This is exacerbated when there is more than one reduction alternative. For instance, if the OS decides that the streaming application is spending too much energy, it could throttle the allocation of CPU or network resources. Yet, the OS cannot decide which resource restriction will produce the most acceptable degradation.

2. Applications are not always aware of user priorities. Applications are in a better position than the OS to prioritize actions, yet they may still need user input to avoid dissatisfaction. As highlighted in the streaming example, an application might not have a total order of its configuration sets. Consequently, the application developer cannot determine the utility levels of the Pareto frontier for a given user. Being so, it is only the user who can determine the relative value of each functionality bundle. Just knowing the user's desired battery lifetime is not sufficient to automatically maximize utility.

3. Users should choose at the right level, trading off functionality versus lifetime. Although existing energy-management systems involve users, many require too much knowledge from users at the wrong level of abstraction. Users should only care about high-level functionality and device lifetime, instead of worrying about what phone subsystem affects how much battery drain. A user expecting her battery to survive a 12-hour flight should not need to tinker with the screen brightness, CPU frequency, scheduling algorithm, or the WiFi data rate of her smartphone. The device should hide these tradeoffs whenever possible. Popular solutions for end-user energy management are based on *components* rather than functionality, requiring users to know the resource implications of turning off the radio connection, GPS, network synchronization, or Bluetooth. Frameworks like Cinder [198], Koala [215], and Chameleon [152] have mechanisms to limit resource usage per



(a) (b) (c) (d)

Figure 3.4: *Modes* abstract complex settings using a single functionality-centered concept and are common in diverse occasions: (a) Airplane mode for smartphones, (b) Incognito mode in the Chrome browser, (c) Scene modes for a camera, and (d) Driving modes for a vehicle's semi-automatic transmission.

application but suffer from the same problem – they assume that mobile users are likely to become system administrators or programmers of their own devices. On the other hand, other frameworks limit themselves to a single knob, such as lifetime [104, 243] or a preference to performance or duration [38]. We show that in some cases this is not enough to maximize utility.

The only remaining question is why the OS should be involved at all, since applications could directly elicit preferences from users. The challenge lies in deciding when apps should offer choices. Such decision requires knowledge of current and future energy availability, which naturally resides in the OS level [75]. If each application had its own battery-prediction mechanism, then developer effort would duplicate, thereby leading to poor or inconsistent behavior. The OS, on the other hand, is in the right position to provide energy context to all applications.

3.4 Application Modes

To address the concerns mentioned in the previous section, we have developed and implemented a new abstraction named *Application Modes* (AM) – bundles of functionality that are declared by applications to ease the separation of roles between applications, users, and the OS to enable effective resource management. We borrowed the concept of Modes from several commonplace scenarios (see Figure 3.4), where each presents complex settings abstracted as a single, functionality-centered and easy-to-understand concept. Application Modes resemble Chroma [48] in that very little application knowledge is exposed. Different from Chroma, users are not oblivious to the system's arbitration but actually have an active voice in the decisions that affect their experience. Application Modes are particularly well suited to cases where a user's implicit-preference

function involve multiple dimensions, with no total order among them, similar to the different shooting modes of a camera, for example.

Power savings are achieved via *graceful degradation* of individual applications. Developers devise different sets of user-perceived functionality in exchange for different levels of energy consumption. Various paths lead to graceful degradation: different settings, different algorithms [201], even different programs. It is up to the developer to create those sets after assessing their energy impact. Most commonly, developers will choose strategies that, although different in implementation and QoS, lead to the same application goal.

Applications Modes offer a preferable granularity at which the OS can profile energy consumption and make lifetime predictions. In parallel, applications keep control of *what* resources to use so that the device draws less power, but leave the decision of *when* to do so to the OS, which has detailed knowledge of the energy context of the entire device, and to the user, who can prioritize functionality based on demand.

3.5 Design and Implementation

We have implemented the AM framework atop the Android OS. We chose Android as our testing platform because of its open-source nature and current mobile-market dominance [125]. Our implementation does not require device rooting nor kernel modifications, making it easily accessible to the majority of Android users.

Four modules comprise our system; these modules communicate with each other to enable informed decisions on fine-grained power management. Figure 3.5 illustrates AM's components and how they are connected. We explain how each component works in the following sections.

3.5.1 Developer API

Most users are not interested in the internals of software development, hence developers willing to distribute bundles of functionality for a target application should export mode metadata that are relevant to end users. These include: (1) mode label for identification; and (2) mode description so that users can qualitatively assess the impact of each functionality choice. The API exposes the developer-defined modes of a particular

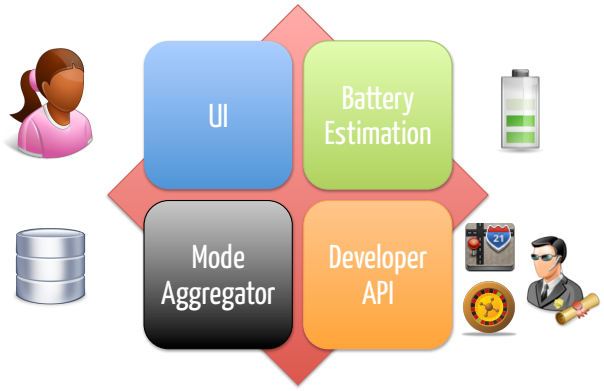


Figure 3.5: System components of the AM framework. Developers define bundles of functionality using the provided API. Bundles are presented to users along with battery-lifetime estimates from the OS.

application to the entire system so that users and the OS can assess them.

The API consists of a narrow interface between applications and the OS (cf. Listing 3.1). When opened for the first time, applications linking to this interface via a shared library declare to the OS each implemented mode using the attributed label and description.

```
registerModes(List<ModeData>);  // system call

setMode(ModeId);                      // callback
```

Listing 3.1: Modes API for data exchange between applications and the OS.

3.5.2 Mode Aggregator

The Aggregator includes a daemon (`RequestResponder`) that intercepts mode-registration requests and immediately registers the application and its mode metadata using a series of IPC-message exchanges onto a local database. If the supporting application is erased, the aggregator garbage-collects the corresponding mode metadata. Supported applications promise to switch between modes when instructed by the OS, whereas applications oblivious to the new API are not affected by it. Table 3.2 lists examples of representative Modes for different applications.

Application	High	Medium	Low
Pedestrian Tracking	1m precision, real-time tracking	100m precision, 15min update interval	500m precision, updates at least once every hour
Car Navigation	3D map, audio, real-time location	3D map, audio, location update near turns	2D with static directions
Media Streaming	HD video, real-time streaming	SD video, real-time streaming	SD video, upload while charging
Twitter	Real-time post updates	Updates every 30 minutes	Updates on demand

Table 3.2: Mode examples for some representative mobile applications.

3.5.3 Battery Life Estimator

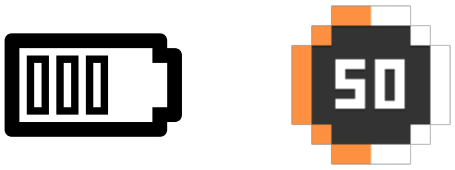
The amount of remaining battery charge can substantially impact how users prioritize their interaction with handheld applications and when users will recharge their devices. To effectively select on which mode an application should run, users need feedback on the impact of their choices over the battery-discharging behavior. Users commonly take notice of the remaining battery life from an interface that displays the battery's state of charge using level bars or percentage (Figure 3.6). Although more user-friendly than voltage or Coloumb (mAh) numbers, the meter interface does not clearly present an estimation of how long the battery will last. It is tempting to assume a linear relationship between the battery percentage and the remaining battery time. However, such assumption does not always hold for different reasons:

- A dynamic load can heavily influence the battery-discharge rate. Due to a nonlinear physical effect on the battery, the lifetime depends on the usage pattern. During periods of high energy consumption the effective battery capacity degrades, thus shortening its lifetime. Conversely, during periods of low energy consumption the battery can recover some of its lost capacity, thus lengthening its lifetime [54].

- The temperature within the battery may also vary due to a dynamic load or an external heat or cold source. The energy consumption of a battery-powered system is impacted by the internal resistance of the battery, which varies from device to device [91]. The resistance strongly depends on the temperature – for Li-ion batteries, the resistance increases about 1.5× for every 10°C of temperature decrease [54].

Since predicting battery lifetime is hard, handheld systems restrict themselves to low-resolution metering interfaces and users usually resort to guesswork and previous experience to determine how much battery life they have and how they should prioritize apps.

Nonetheless, as we naturally plan ahead and prioritize tasks using the concept of deadlines, time is the



(a) (b)

Figure 3.6: UI widgets for smartphones commonly used to inform users on the battery's state of charge: (a) bars; (b) percentage.

most natural way to express how long batteries last. Just like other works [191, 221], we opt to use time as a proxy to combine user interaction with power management. Instead of trying to predict future lifetime, we pre-compute it using a history of measurements taken from representative workloads.

3.5.4 User Interface

The user interface is the front end to the Aggregator and presents to the user the available modes of operation for the applications that leverage the AM API. Besides displaying each mode's metadata, the interface also presents an estimate of the battery-lifetime reduction (or increase) for each mode. From the quantitative (lifetime) and qualitative (description) data, users have a better understanding of the tradeoffs between performance and battery duration than what is currently offered in commercial OSes. Once a mode is selected, three steps take place.

1. The Modes UI notifies the system of a mode switch. Based on the available metadata, `RequestResponder` routes the mode selection to its application owner.

2. The enquired application modifies its functionality by switching its internals to another set of developer-defined changes.

3. `RequestResponder` updates its application database with metadata on the applied changes.

If we consider the scenario of running multiple AM-supporting applications concurrently, configuring each application becomes a repetitive and onerous task. The AM UI has a second course of operation in which

all supported applications are listed, descendingly sorted by battery-lifetime increase. Users can then prioritize and configure only those applications that matter the most to them.

3.5.5 Putting Everything Together – Workflow

Application Modes represent a meaningful granularity at which the OS profiles energy consumption and predicts battery lifetime. Applications keep control of what resources to reduce in order to save energy, but leave the decision of *when* to do so to the OS and to the user.

In one possible scenario, the OS notices that at the current power draw, the device will exhaust the battery before the next expected recharge. The OS reacts by presenting to the user a list of running apps, ordered by battery impact. When the user selects an app from the list, the OS presents an interface similar to that in Figure 3.7. From this interface, the user selects a different mode of operation, informed by its description (i.e., functionality) and expected impact on battery lifetime. In another use case, the user is presented with a notification of AM support when opening a newly installed application. After clicking the notification, the aforementioned interface appears, and the user can explore the tradeoff possibilities offered by the application developer. Once a mode is selected, the OS instructs the application to change its settings. It is the responsibility of the application developer to instruct her software to change its behavior according to the mode selection.

3.6 Evaluation

We evaluate Application Modes in two scenarios: navigation and gaming. For each scenario, we consider three evaluation metrics: (1) development cost (in lines of code); (2) average power draw; and (3) battery lifetime, measured from continuously running each scenario from a full battery to its depletion.

Navigation

Over the past two decades, devices used to navigate drivers and pedestrians have proliferated, with smartphones being the most versatile. Thanks to their multifunctionality, price, and accurate position sensors,

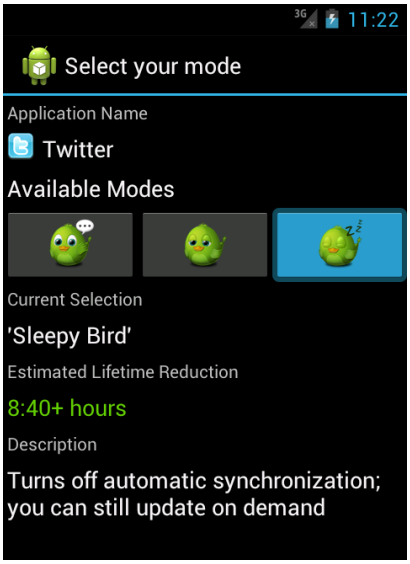


Figure 3.7: Interface to select application modes for the Twitter mobile app.

smartphones and tablets have taken over the navigation market that once belonged to dedicated GPS systems [230].

Turn-by-turn navigation exercises several hardware subsystems, including the CPU, GPU, audio, network, and GPS. It is sometimes used in critical situations, when there is little battery left and the user needs orientation to arrive at her destination (and a charging opportunity). There are also interesting tradeoffs in functionality, utility, and energy use, depending on what subset of resources the navigation app uses.

We demonstrate the potential savings in navigation by evaluating the energy output of different modes for OsmAnd [22], an open-source Android navigation app with online and offline features. We consider five modes, listed in Table 3.3, from select parameters for screen and audio output, map-data source, and routing mechanism. These settings are not transparent to the user, and offer specific tradeoffs between location accuracy and resource use for a given route. Notable differences between settings include the use of previously downloaded vector maps instead of streamed tile data, disabling the display and using audio for directions, and only displaying directions for the user to write down.

Defining the five new modes required a total of 129 additional lines of Java code. It is important to note that we refined these five modes after extensively studying OsmAnd's source code, from which we discovered

Mode	Display Settings	Routing Engine	Audio?
High Quality	Online map tiles and overlays, PoIs, compass display, 3D view	Cloudmade (online)	Yes
Light Screen	Offline vector maps, no PoIs, no compass, polygons, day mode (light screen)	OsmAnd (offline)	Yes
Dark Screen	Offline vector maps, no PoIs, no compass, 2D view, night mode (dark screen)	OsmAnd (offline)	Yes
Audio Only	Screen off	OsmAnd (offline)	Yes
Written Directions	Screen on for directions search, off afterwards	Cloudmade (online)	No

Table 3.3: Mode alternatives to a navigation app guiding a four-mile trip. Upper modes yield higher-quality routes in exchange for greater resource usage.

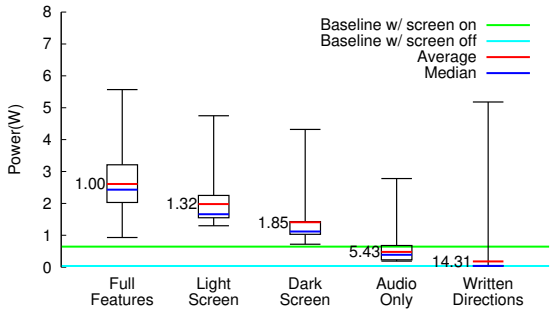


Figure 3.8: Power-draw distribution of different profiles OsmAnd (cf. Table **??**). Box limits represent the first and third quartiles and include the median and average power draw. Whiskers indicate the minimum and maximum power draw. Numbers to the left of each box show the improvement in battery drain relative to "High Quality." Reducing the output quality allows for greater battery savings.

26 independent settings that could affect both functionality and energy consumption.

For the sake of comparison, we measured the power draw of each mode during a fixed, four-mile car trip. Figure 3.8 depicts the distribution of instantaneous power-draw samples for our device under test (Samsung Galaxy Nexus). The "Full Features" mode yields a richer visual trajectory, including information such as points of interest (PoI), at the expense of a higher power profile. As we reduce the number of enabled settings, there is a constant drop in energy consumption along with a decrease in the perceived quality of the routing information. "Written Directions" draws on average 14× less power than "Full Features." In return, the user has to take notes of the route before the trip starts and use them as the only navigation source to reach her destination[1].

Mode	Game Settings
High	640x400 resolution (scaled), 100% screen brightness, HW acceleration, music and SFX enabled
Medium	160x100 resolution (scaled), 50% screen brightness, HW acceleration, no music, SFX enabled
Low	320x200 resolution (no scaling), 50% screen brightness, no HW accel, no music, no SFX

Table 3.4: Mode alternatives for DOOM. Upper modes yield a more immersive experience.

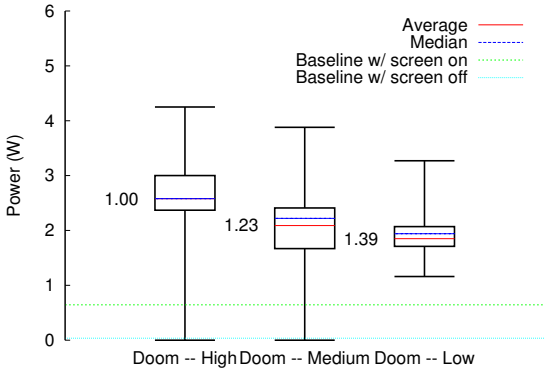


Figure 3.9: Power draw of different modes for DOOM (game). Lower settings lead to greater battery savings at the expense of a less smooth audio and visual experience.

Gaming

The rise of smartphones and tablets has ushered a renaissance in gaming. Games have been used as a benchmark for performance and resource use for years and are one of the main sources of battery depletion [126]. We analyze how AM can help mobile gamers enjoy a longer battery lifetime by using the first-person-shooter game DOOM as a case study. We implemented three different modes (Table 3.4) for PrBoom [24], the open-source port of DOOM to the Android OS, totaling extra 119 lines of Java code.

Figure 3.9 shows the power draw of the three proposed modes. "Low" caters to users who want to extend their gaming session despite a low battery. To reduce the battery-discharge rate, "Low" reduces the screen resolution without scaling it. As a result, the display output is shown on a small window surrounded by a large black frame. Given that the Galaxy Nexus smartphone has an AMOLED screen, the black frame along with the lower brightness setting significantly reduce the screen's power draw (72% less than the high quality mode). Figure 3.10 depicts the measured battery lifetime of each mode.

[1]This mode was motivated by real experience!

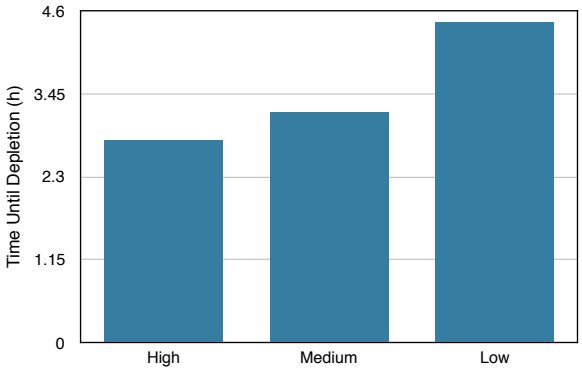


Figure 3.10: Battery lifetime for DOOM's application modes.

3.7 Challenges, Limitations, and Extensions

During the development and evaluation of Application Modes, we faced a variety of challenges and open questions that defined potential steps for further research, including:

Conflict Detection and Resolution. As the number of applications supporting AM increases, there is a chance of conflicting choices when configuring applications that share subsystems. For instance, if two apps use the radio simultaneously, having only one app commit to reduce its networking is of little use. The OS should have a mechanism to detect and resolve such conflicts, with possible involvement of applications, and, ultimately, end users. Another source of conflicts are global settings not associated with a particular application – screen brightness and Airplane mode being major examples. A possible solution is to track action intents [96, 99] that lead to energy changes so that the system can reach an informed resolution.

Engaging Users. Reeves and Nass [193] argue that people behave towards and expect of computers in ways consistent with human-to-human contact and relationships. There is an expectation that technology should treat us humanely and act with human values as soon as computers show the slightest sophistication. Therefore, users will only see value in Application Modes if the framework converses in a language that meets user standards. Our design decisions partially cover the semantic gap between what users expect and what machines can provide. Nonetheless, more research is necessary to create an adequate battery-handling interface that can quickly adapt to ever-changing user needs and contexts. Although we initially pursued user and developer

engagement in power management, our focus now mainly switches to the latter.

Engaging Developers. Optimizing software for energy efficiency is an challenging task. In the case of application-specific optimizations, deep analysis and debugging are necessary. Therefore, application developers need a means that can assist or automate the search for energy hotspots. The remainder of this dissertation focuses on techniques that contribute to this objective.

3.8 Summary

Application Modes promote the cooperation of the OS, applications, and users to achieve effective energy management in mobile devices. Applications provide the OS with discrete modes that express graceful degradation in face of limited battery. The OS centralizes all of the power-management knowledge facilitating the separation of roles and easing the lives of users and developers. We implemented our prototype atop the Android OS and demonstrated the battery-saving potential beyond energy proportionality of three representative mobile applications. A major challenge is the extra burden on the developer's side, who is now also responsible for discovering appropriate functionality changes and analyzing the corresponding impact on battery life. In the following chapters, we elaborate on semi-automatic techniques that guide developers towards the most promising energy-saving opportunities beyond energy proportionality.

Chapter 4

TAMER

Generalizing the Exposure of Software-Related Energy Knobs

Application Modes is our first attempt to control and modify application functionality for the benefit of battery savings. A major caveat of AM is the requirement of point solutions – each application must be recoded and redistributed before users can re-evaluate battery management. Not all developers are willing to spend time doing additional work without a guarantee of better results. Therefore, we must scale out the idea of functionality change by avoiding this extra burden. In this chapter, we consider the idea of rewiring an application binary to achieve energy savings. We develop a system capable of quickly modifying the behavior of applications running in the background.

4.1 Introduction

Due to its user-centric and interactive nature, the flow of a mobile application is driven by events such as user actions, sensor I/O, and message exchanges. Such event-driven paradigm lets the system idle until a new event arrives. Mobile OSes, such as Android, iOS, and Windows Phone, take advantage of idling opportunities to engage in *opportunistic suspend*. Upon brief periods of idling, the handheld returns to the default suspend

state. Hardware subsystems, including the CPU, GPU, GPS, and network modem, shift to a low-power mode and software state is temporarily stored in self-refreshing RAM. The same subsystems return from suspension upon interrupts emitted by hardware or software indicating that there are pending requests.

With the rise of multitasking and the multiplication of background services and complex mobile applications, we expect the amount of interrupts to increase, thereby forcing the system to spend more time *active* attending requests. Such active periods take a heavy toll on battery lifetime. A recent study by Google quantifies this impact: each second of active use of a typical smartphone reduces the standby time by two minutes [163].

This chapter studies the problem of battery drain mostly due to app-originated background operations that wake up the mobile device. We present Tamer, a mechanism we built for the Android OS that interposes on events and signals responsible for task wakeups – alarms, wakelocks, broadcast receivers, and services. Like a number of profiling tools, Tamer lets developers characterize the background behavior of different apps installed on a device. In §4.2, using Tamer's instrumentation, we show how a set of installed applications can dramatically affect the battery lifetime of four different devices. Unlike existing profiling tools, however, Tamer can also selectively block or rate-limit the handling of such events and signals according to flexible policies. Tamer controls the frequency at which software schedules alarms or receives notifications of specified events, thereby providing fine-grained control over the energy consumption of apps that are useful yet irresponsible or inefficient with respect to background activities. In §4.5.2 we show via a few examples how Tamer can reduce the battery drain due to energy bugs [178].

We summarize our contributions as follows:

- We characterize how applications and core components of the Android OS use specific features to perform background computing, and how background computing significantly affects energy consumption. In special, we note that Google Mobile Services play a major role on battery drain while the device is apparently dormant (§4.2).
- We introduce Tamer, an OS mechanism to control the frequency at which background tasks are handled thereby limiting their impact on energy consumption (§4.4). Tamer leverages code-injection technology and is applicable to any Java-based Android application.

- We demonstrate how TAMER can successfully throttle the background behavior of popular applications, thus reducing their energy footprint (§4.5). We show how different policies reduce power draw in exchange for little visible impact on functionality.

Despite being a powerful mechanism, TAMER is only a step towards effective energy management. In §4.6, we discuss challenges in helping developers define policies that are effective yet not disruptive to the user experience.

4.2 Motivation

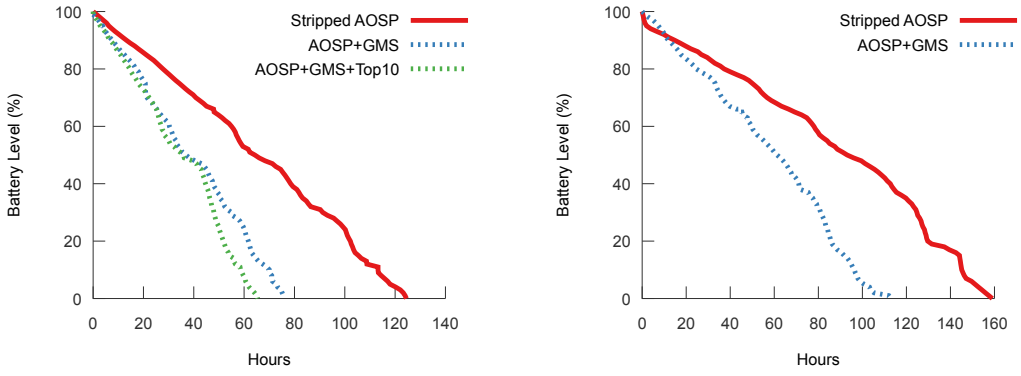
Device Name	Device Type	Processor	Features
Google Galaxy Nexus	Smartphone	Dual-core 1.2GHz ARM Cortex-A9	WiFi, GPS, 3G (off)
Samsung Galaxy S4	Smartphone	Quad-core 1.9GHz Qualcomm Krait 300	WiFi, GPS, LTE (off)
Amazon Kindle Fire 2	Tablet	Dual-core 1.2GHz ARM Cortex-A9	WiFi
ASUS MeMO Pad 7 (ME176C)	Tablet	Quad-core 1.83GHz Intel Atom Z3560	WiFi, GPS

Table 4.1: List of devices under test (full battery drop).

Many smartphone and tablet users are habituated to an always-constant Internet connection, which is necessary for immediate notifications of e-mails or application updates. Other common background operations include polling navigational sensors for location clues and turning on the network radio for incoming messages. There is little restriction in Android on what apps can do in the background and developer's discipline is the only barrier preventing inefficient apps from hogging resources and wasting energy.

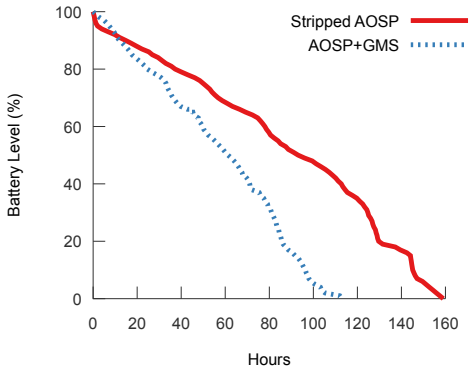
Traditionally there has been little visibility, both to app developers and to users, on the contribution of individual apps to energy use, especially while in the background. It is also challenging to visually recognize whether an application is running or idling while in the background. Given that one cannot optimize what cannot be measured, recent monitoring and profiling tools have helped bridge this visibility gap [45, 111, 167, 174, 179, 189, 223].

Today's average handheld includes a large amount of third-party software. According to a recent Nielsen report [79], the typical US smartphone owner has around 27 apps installed on her device. Even with the best available tools, the end user can do little to cope with inefficient apps. Most of the aforementioned tools target

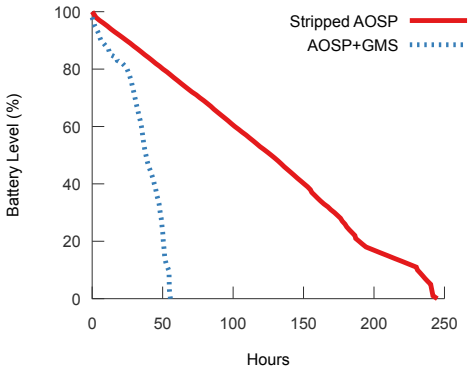


(a) Galaxy Nexus

(b) Galaxy S4



(c) Fire 2



(d) MeMO Pad 7

Figure 4.1: Battery drop of four Android devices idling with the screen off. By adding Google Mobile Services to the base installation, the relative battery lifetime decreases between 29.5% (Fire 2) and 77.5% (MeMO Pad 7).

developers and, even so, are coarse grained. For instance, eStar [167] and Carat [174] can identify the most energy-inefficient program, but only offer to kill or uninstall the culprit app, perhaps suggesting replacements. This palliative solution does not work for apps that exclusively provide irreplaceable functionality.

TAMER offers the possibility of a much finer-gained control once an energy hog or bug is found. TAMER hints on tasks expending the most energy and can rate-limit their execution. TAMER does so by tracking most of the events responsible for device wakeups that are visible at the Android-OS framework level, and can filter their continuation in real time.

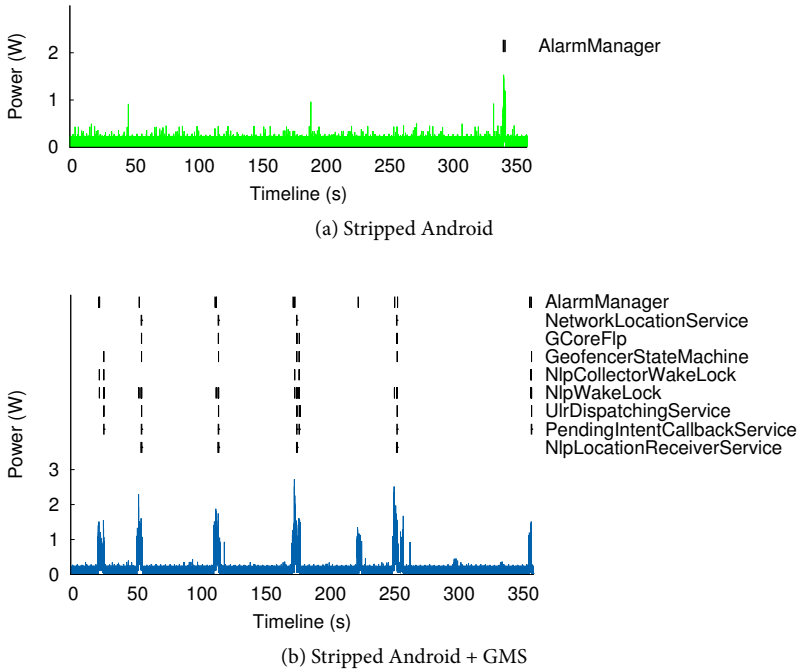
To demonstrate how running tasks can significantly impact battery life, we measured the battery drop of four Android devices (two smartphones and two tablets, cf. Table 4.1) running two different application sets, *while idling and with the screen off*. We consider three scenarios: the first testing environment ("Stripped Android") consists of a stripped version of the Android OS (KitKat), containing a minimum number of services and apps; the second environment ("Stripped Android + GMS") adds Google Mobile Services (GMS) on top of "Stripped Android." GMS consists of proprietary applications and services developed by Google, such as Calendar, Google+ (social media), Google Now (personal assistant), Hangouts (instant messaging), Maps, Photos, Play Service (integrating API), Play Store, and Search. Due to their popularity and added value, GMS apps are included in most of the Android devices sold today. For the third scenario, which we ran on the Galaxy Nexus smartphone, we took the GMS install base and added the most popular free apps from Google's Play Store as of January 2015[1].

For each experiment, we left each of the four devices unattended and configured to its default settings. Relevant settings include a established connection to a WiFi access point, enabled location reports via GPS and background network synchronization. We expect most of the battery drain to stem from static-voltage leakage and eventual background processing.

Figure 4.1 shows the time taken by each environment-device combination to deplete its battery. For all devices, "Stripped Android" took the longest to completely drain the battery. For both tablets evaluated, the

[1]Crossy Road, Candy Crush Saga, Pandroid Radio, Trivia Crack, Snapchat, Facebook Messenger, Facebook, 360 Security Antivirus, Instagram, and Super-Bright LED Flashlight.

draining-time difference spanned dozens of hours. To investigate the motive behind the large difference between depletion times, we instrumented the Android software stack to timestamp the occurrence of background events. Additionally, we connected one of the four devices (Galaxy Nexus) to a Monsoon power monitor and collected power samples from its battery. Finally, we aligned and synchronized both the event and power timelines to understand their correlation. Figure 4.2 depicts a six-minute slice of this combination. We observe that GMS triggered more events in the background and that they are correlated with the surge of power peaks. We used this tracing knowledge to build a mechanisms that counters the energy buildup of excessive wakeups. Because this mechanism relies on Android's internals, we first need to understand how an Android application functions while in the background.



(a) Stripped Android

(b) Stripped Android + GMS

Figure 4.2: Event tracing and power measurement on the Galaxy Nexus smartphone for two scenarios. For each graph, the top stack shows the event calls over time. The bottom curve depicts the corresponding system power draw during the same period.

4.3 Android OS: Power Management and Application Internals

This section provides a concise description of Android's power-management system followed by an overview of the components of a typical mobile application and how these components behave while running in the background. Finally, we highlight the influence of background execution on battery drain using four types of events: wakelocks, services, broadcast receivers, and alarms. §4.4 describes TAMER, a control system that adjusts the frequency at which these events are dispatched.

4.3.1 Mobile Power Management

Android KitKat employs an aggressive form of power management to extend battery life. By default, the entire system suspends itself, sometimes even when there are running processes. Opportunistic suspend is effective in preventing programs from keeping the system awake and quickly draining the battery. To curb system suspension, Android uses `Wakelocks` to keep the system awake. Wakelocks are reference-counted objects that can be acquired and released by kernel and userspace code. A wakelock acquire expresses a process's need for the system to remain awake until run-completion. A wakelock acquire either holds a resource awake until there is a corresponding release call, or sets up a timer to relinquish the lock.

Kernel drivers use wakelocks to block the suspension of different subsystems (e.g., CPU, network, and screen), whereas the Android application framework leverages wakelocks for different levels of suspension, represented by groups of components (e.g., keep the network radio awake vs. keep the radio, screen, and CPU awake). As an example of suspend-blocking by the OS, Android automatically acquires a wakelock as soon as it is notified of an input event and only releases the same wakelock once some application handles the event or there is a timeout. Application developers can also instantiate and manipulate wakelocks using the Wakelock API. An e-book reader app must acquire a wakelock to keep the screen awake so that the user can read her favorite novel without interruption. Wakelocks play an important role in guaranteeing proper background task execution in face of default suspension, as we will see next.

4.3.2 Android Applications: Dealing with Lifecycle Changes

Barring a few interface-less system services, an Android application consists of a set of `Activities` that places the UI widgets on the screen. An application starts with a single thread of execution attached to the *foreground* UI, which is mostly responsible for dispatching interface events. To avoid unresponsiveness and user frustration, a wise programmer would offload other computations to concurrent worker threads while the UI responds to input events. Support for concurrency exists in the form of a number of standard Java classes and interfaces, such as `Threads`, `Runnables`, and `Executors`, as well as Android's own flavors – `AsyncTasks` and message `Handlers`. However, such primitives cannot wake up the system from suspension.

As the user navigates through, out of, and back to an application, its lifecycle switches between different states according to the UI visibility. An application is active if one of its activities receives user focus in the foreground. If the user switches to another app or decides to lock the device, the application is paused and moved to the background. Because mobile apps are multitasked, developers need ways to run tasks even when her app is not occupying the screen.

Dispatching Background Tasks

The small screen size of a smartphone or tablet prevents multiple applications from running simultaneously[2]. To conserve energy, backgrounded apps are frozen and stop running either as soon as another application is brought to the foreground or if the screen is locked. Context switching enables opportunistic suspend – by making apps invisible, Android frees its own set of wakelocks, opening space for hardware throttling.

Android offers to developers a narrow and well-defined interface for background-task offloading that takes care of scheduling latent tasks [39]. This interface consists of a handful of components including services, broadcast receivers, and alarms. The internal implementation of these components leverages wakelocks to keep the device awake while executing tasks.

[2]There are a few attempts to support shared-screen applications, although they are far from the norm.

`Services` are application components that asynchronously run on background threads or processes. `Activities` dispatch services to perform long-running operations or to access resources on behalf of applications, such as synchronizing local data with a remote storage system. An advantage of running a separate service is that its execution persists regardless of the state of its owner's UI.

A `BroadcastReceiver` is a reactive mechanism that permits programs to asynchronously respond to specified events. An application registers a `BroadcastReceiver` along with an event-subscription list, the `IntentFilter`, which is used to determine whether an application is eligible to respond to a given event. Events are predefined by the system (e.g., "battery charged") or can be defined by developers (e.g., "backup finished"). Receiver threads remain dormant until a matched event arrives; they respond by running a callback function. A file-hosting app could, for instance, register a receiver to display a notification once a scheduled backup finishes.

Another common programming pattern is the ability to perform time-based operations outside the lifecycle of an app. For instance, checking for incoming e-mails every so often is a recurrent user operation that can be automatized. Developers use the `AlarmManager` mechanism available in the Android SDK to schedule periodic tasks at set points. At each alarm trigger, the system wakes up and executes a callback function whose contents can take various forms: a UI update, a service dispatch, an I/O operation, scheduling a new alarm, etc. Alarms are a good fit for opportunistic suspend – apps are only activated when there is pending work.

In summary, Android KitKat provides three types of asynchronous mechanisms to run background tasks: services, broadcast receivers, and alarms. Aligned with wakelocks, developers have a powerful collection of events that can keep the system awake. In the next section, we introduce Tamer, a system that acts on this narrow and well-defined interface to throttle the rate at which background events are handled in exchange for energy savings.

4.4 Tamer

4.4.1 Design

Given that background events can affect the sleeping pattern of mobile devices, we consider the possibility of regulating the frequency of these events to improve battery life. We introduce a policy-and-control mechanism to regulate the interval between event dispatches. We model this regulatory process using three steps: (1) observation; (2) comparison; and (3) action.

First, we establish a policy mechanism that allows developers and savvy users to declare how often the running system should let a background event proceed. A policy is a contract that declares the conditions for event execution. This contract specifies the event type and its identifier; an optional list of affected apps, if the restriction only applies to a subset of event dispatchers or receivers; whether the policy enforcement should also take place when the event owner (application) is running in the foreground; and the rate at which the event is allowed to execute. A developer could, for instance, install a policy for a weather-forecast app that accepts calls to `WeatherUpdateService` at most once every six hours, whereas calls to `LocationUpdateService` from the same app would remain unlimited.

To enforce policies, we outline a controller comprising three agents: observer, arbiter, and actuator. The *observer* intercepts event occurrences and bookkeeps their frequency. The *arbiter* verifies whether the measured event rate is above the policy-declared threshold, in case it exists, and notifies the *actuator*, which hijacks the event continuation to artificially reduce its occurrence rate. Figure 4.3 illustrates Tamer's control sequence.

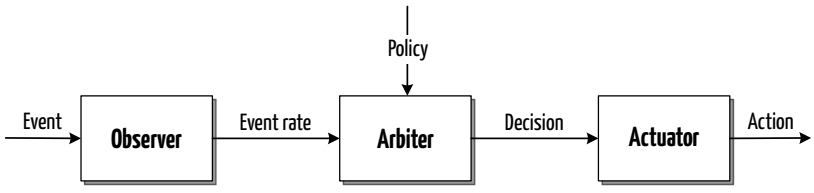


Figure 4.3: Sketch of Tamer's event-control system as a three-stage pipeline.

There are two ways to fulfill event throttling: canceling or delaying. An event cancel denies the continuation of the event payload by early returning from the callback function, thereby preventing its complete execution.

An event delay, on the other hand, postpone the continuation of the event call for a limited time. In this chapter, we consider the canceling strategy. We discuss the pros and cons of each strategy in §4.6. It is important to note that an event cancel does not lead to a program crash. Alarms, services, and broadcast receivers all run asynchronously. Wakelock requests are synchronous, although a run-denial greenlights system suspension when there are tasks expecting to run. Still, these denied tasks are never aborted, but run in chunks as long as the system periodically wakes up. Our approach tries its best to prevent the scheduling of unwanted events. When that is not possible, TAMER *aborts* the event continuation at the earliest opportunity to avoid the energy cost of the event payload.

To drive the implementation of TAMER, we established the following requisites:

Comprehensive support of events. TAMER's control mechanism should be inclusive. Although app-specific solutions are effective, they seldom apply to other programs. TAMER's control sequence should monitor and, if necessary, actuate on every background event instance. Details about a particular event should be confined to its policy and not affect the controller. Furthermore, the policy designer should be responsible for defining a sane event frequency, considering, perhaps, the context and the impact of an event hijack. To implement an all-encompassing monitor system, we opt for an OS-level solution.

Support for power-oblivious applications. Users should not abstain from using their favorite apps even when these apps are power hogs. Uninstalling or suggesting alternative apps for the purpose of saving energy is not acceptable. TAMER should cope with the existence of ill-behaved apps and act upon their misbehavior if directed by the policy designer.

Compatibility. Solutions that rely on deep system introspection require extensive rewrites of components [41, 88, 96] or even development from scratch [198]. Although tempting, straying from the mainline can limit the user base, especially in the case of consumer-oriented OSes. Developers interested in controlling and analyzing system behavior need an environment that resembles as much as possible the what is available to users. TAMER should be compatible with and keep a minimum amount of changes to the existing underlying OS.

Efficiency. Mobile apps must cope with limited computational and energy resources. The control mechanism should avoid high computing overhead to prevent excessive battery drain and system slowdown.

Event	Class	Hook Point (Method)	Instrumentation Payload
Wakelock	com.android.server.PowerManagerService	acquireWakeLockInternal	If there is a matching policy, early return in case acquire happens before grace period. Else, let acquire proceed; bookkeep wakelock name and start grace-period timer.
		releaseWakeLockInternal	Called only if acquire was not blocked; log wakelock-hold interval.
Service	com.android.server.am.ActiveServices	startServiceLocked	If there is a matching policy, proceed as in wakelock acquire.
BroadcastReceiver	android.app.ContextImpl	registerReceiverInternal	Save pointer to receiver declared at runtime.
		unregisterReceiverInternal	Remove pointer to receiver declared at runtime.
	com.android.server.pm.PackageManagerService	addActivity	Save pointer to receiver declared at compile time.
		removeActivity	Remove pointer to receiver declared at compile time.
	com.android.server.am.ActivityManagerService	broadcastIntentLocked	If there is a matching policy, temporarily undeclare receiver to prevent event broadcasts; update grace period.
Alarm	com.android.server.AlarmManagerService	triggerAlarmsLocked	If there is a policy, proceed as in wakelock acquire.

Table 4.2: Tamer's instrumentation points.

4.4.2 Implementation

To avoid reimplementing OS components to regulate event handling, Tamer uses the Xposed framework [34] to enable system modifications at runtime. Its main caveat is the need for device rooting. Xposed enables thorough system modifications without need for binary decompilation. Xposed intercepts Java method calls and temporarily diverts the execution flow to function-hook callbacks for inspection and modification. Developers define callbacks as separate modules that run in the context of the intercepted application. Function hooking works by matching the method's name and signature of the declared callback with the currently running function. Xposed allows for changing the parameters of a method call, modifying its return value or terminating it early. We use this hooking mechanism to intercept function calls originating from or directed to the aforementioned background event types (i.e., wakelocks, services, alarms, and broadcast receivers). Hook modules are distributed as independent Android apps and are not bound to a specific Android version, being compatible with the majority of customized Android releases.

Figure 4.4 shows how Tamer communicates with the Android OS. Tamer sits, along with Xposed, between user applications and the Java-based application framework, which serves as the foundation for the Android SDK. Events have directions that help define how to write the interception payload. While service and wakelock

calls originate from apps and are forwarded to the framework, alarms and broadcast receivers work in the opposite direction.

Tamer consists of a series of function hooks that interpose on the background-processing interface and acts as a controller mechanism to enforce user-defined policies. To implement Tamer's event canceling, we use Xposed's introspection API to explore, monitor, intercept, and modify public and private classes, methods and members of the SDK framework (Table 4.2). To decide where to place the controller's instrumentation points, we studied the source code of the Android framework stack. Treating the relationship between subroutines as a call graph, methods exposed in the public interface are leaf nodes, whereas internal methods are parent nodes. In some occasions, we had to backtrack the call graph to find a proper instrumentation point mainly because (1) the public interface did not offer enough context to feed our monitoring system (e.g., missing event name or unclear caller-callee relationship); (2) in the case of receiving events, it is better to interpose on a call as early as possible to avoid unnecessary operations before cancellation; (3) an event call may have more than one function signature, therefore we looked for a converging parent node. We found an exception to last rule when handling broadcast receivers. Applications can declare receivers at compile time via a `Manifest` file or at runtime using the SDK API. Since the Android framework keeps separate data structures for each case, we had to handle them separately.

Our controller implementation covers all versions of the Android OS ranging from Ice Cream Sandwich to Marshmallow. In a handful of occasions, we resorted to different instrumentation points for a given event, mostly due to small differences in the function signature between OS versions. Because the SDK interface is fairly static, covering future versions of Android should not require major changes.

4.5 Evaluation

We evaluate Tamer in three ways. First, we revisit our motivating scenario (§4.2) and use Tamer to extend the battery life of the GMS-based installation. We then investigate how Tamer can effectively mitigate energy bugs, a system behavior that causes unexpected heavy use of energy not intrinsic to the desired functionality

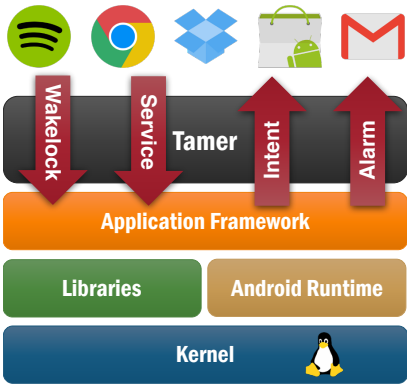


Figure 4.4: TAMER sits between apps and the SDK stack, and interposes on events between these two layers. TAMER is oblivious to the lower system layers.

of an application. Last, we measure TAMER's performance and energy overhead.

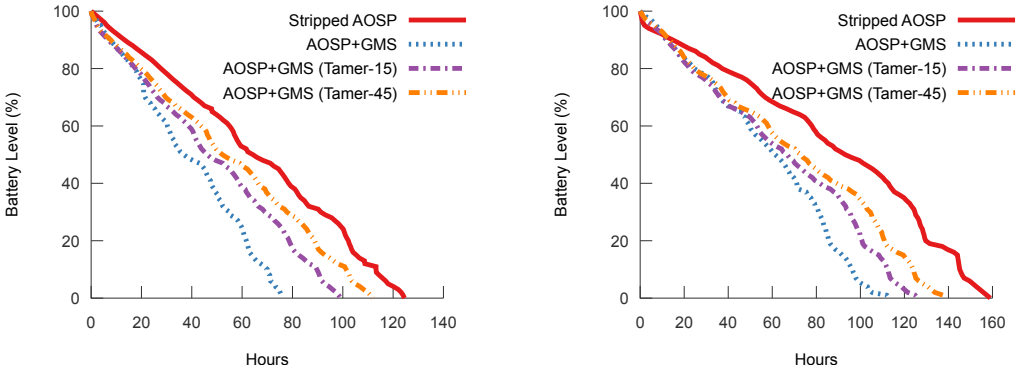
4.5.1 Taming Google Mobile Services

In §4.2, we saw how the inclusion of GMS into the baseline Android significantly reduced the battery life of four devices under test. Nonetheless, GMS adds a series of services and applications that enhances the user's mobile experience. In fact, most users do not even have the option of uninstalling these applications, as GMS comes pre-installed as a system package in the majority of commercial handhelds. We show how TAMER can negotiate a tradeoff between GMS's functionality and battery savings. We aim to keep the added value of GMS without the cost of a silent battery depletion.

Event Name	Type	Count	Total Duration (s)
`NlpWakelock`	W	5963	1662.71
`NlpCollectorWakelock`	W	2121	3926.63
`LocationManagerService`	W	2030	67.12
`NlpLocationReceiverService`	S	1159	-
`NetworkLocationService`	S	579	-

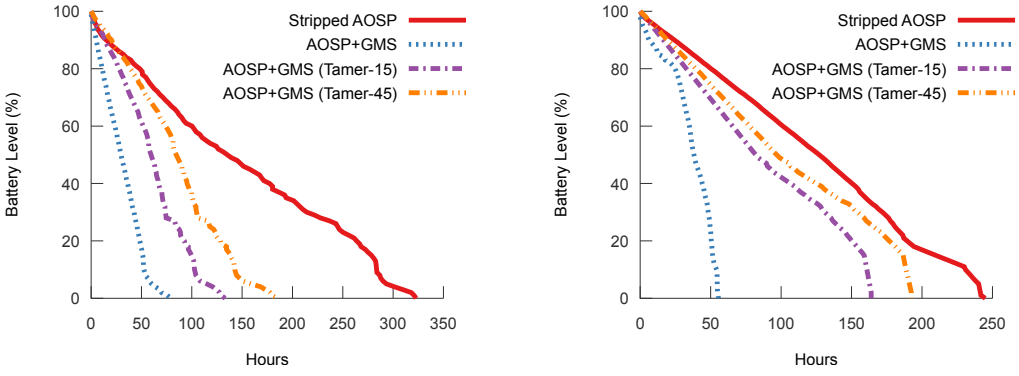
Table 4.3: Top five event occurrences for the Galaxy Nexus due to GMS. A handful of events are responsible for the major impact on the battery. 'W' signifies a wakelock event, whereas 'S' stands for service call.

With the control mechanism established, our next step is to design a policy that reduces the battery impact of events originating from or destined to GMS. Table 4.3 ranks the top triggered events reported by TAMER's monitoring module. For wakelocks, we also report the duration they were held. We use event frequency as an



(a) Galaxy Nexus

(b) Galaxy S4



(c) Fire 2

(d) MeMO Pad 7

Figure 4.5: Battery drop of four different devices after applying our TAMER policies.

heuristic to guide policy configuration. We note that `NlpCollectorWakelock` is primarily responsible for keeping the system awake in the background. Online reports [14, 21] indicate that `NlpCollectorWakelock` is related to *location reporting*, an Android feature used to estimate and report the current device position based on the connection to WiFi access points and cellphone towers. Apps that use this feature include Google Now, Google Maps, among others. The other frequently reported events are also related to the same feature. Disabling Location Reporting on the device's general settings is an easy way to increase battery life, but it turns applications dependent on this feature useless.

The problem with `NlpCollectorWakelock` and other associated events is the frequency at which these events wake up the system and the total time they keep the system awake. For instance, during an 80-hour discharge period of the Galaxy Nexus smartphone, `NlpWakelock` was called, on average, once every minute. In parallel, `NlpCollectorWakelock`, by itself, kept the system awake for more than one hour. We believe such a high battery impact coming from a small set of select applications does not justify the benefits of having GMS running as it is in the background. For this reason, we declared two policies for GMS – Tamer-15 and Tamer-45 – targeting `NlpCollectorWakelock` and its associated events to alleviate their wakeup burden. For each wakelock and service in Table 4.3, Tamer-15 allows a single call of each event every 15 minutes, whereas Tamer-45 allows one event call per 45 minutes. Deciding on an appropriate rate is a subjective matter. Our setup tries to reach a balance between informing subordinate apps of location updates and increasing battery lifetime. Figure 4.5 shows that our policies substantially reduce the battery-drain rate of all four tested devices.

4.5.2 Chasing Energy Bugs

An energy bug, or *ebug*, is a system error either in an application, OS, firmware, or hardware that causes an unexpected, high amount of energy consumption [178]. Such errors occur due to various reasons including programming mistakes, malicious intent, and faulty hardware. Because ebugs usually do not visibly affect application functionality, users will only notice their impact when it is too late – an early emptied battery.

Different from previous research that identified and characterized ebugs [179, 225], we focus on mitigating them at runtime. Tamer allows for testing offending application events at different rates, thus facilitating the

discovery of opportunities for energy savings.

Finding ebugs is not trivial and the lack of a centralized repository of updates samples prevents us from testing our controller more extensively. We present two detailed case studies of new ebugs we found while exploring popular applications for Android. We used eStar [167], a tool that ranks the contribution of mobile apps to battery depletion, to identify our objects of interest. From the handful of applications identified as energy hogs by eStar, we selected two that featured as popular in the following categories of the Google Play Store [13]: Games and Health & Fitness. Although eStar ranks apps in terms of energy inefficiency, we still had to manually verify whether such inefficiency was due to foreground or background execution. For each application, we simulated a user interaction consisting of a short-length active session followed by a long period in the background.

Bejeweled Blitz [7] is an award-winning puzzle game with over 10 million installs from Google Play Store. After a 15-minute play session on the Galaxy S4 smartphone, TAMER reported a single background event by Bejeweled Blitz – an acquire of the `AudioIn` wakelock. Because games are resource-hungry apps, this event call did not instigate initial suspicion. We discovered a red flag, though, after switching Bejeweled Blitz to the background – `AudioIn` unexpectedly remained acquired after the game suspension. To mitigate this bug, we wrote a simple policy that targets Bejeweled Blitz and blocks the renewal of the culprit wakelock during background time. Figure 4.6 depicts the battery drop of a 12-hour session with Bejeweled Blitz loaded in the background before and after applying our TAMER policy. To elucidate the expected battery impact of a well-behaved application, we also measured the battery drop of Candy Crush Saga [9], another famous Android puzzle game that is very similar to Bejeweled Blitz in functionality and user experience. Thanks to our new policy, we see a 4× improvement in battery drain. Figure 4.7 confirms the effect of releasing the ill-behaved wakelock: before being *tamed*, the smartphone CPU cores spent approximately 95% of the time awake. After TAMER's interposition, most of this residency ratio was converted to deep-sleep time.

Nike+ Run Club [20] is a fitness app for tracking runs. This app relies on the GPS sensor, the accelerometer, and the barometer to estimate distance and running speed. According to TAMER, Nike+ Run Club holds five wakelocks while running: `AudioMix`, `FullPower Acc Sensor`, `FullPower Pressure Sensor`,

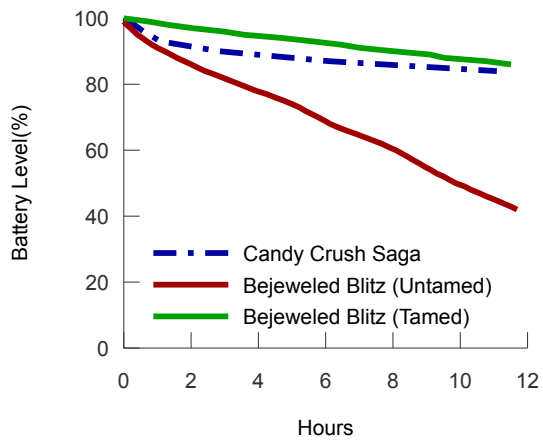


Figure 4.6: Battery drain over 12 hours of Bejeweled Blitz before and after applying a TAMER policy blocking the `AudioIn` wakelock. In both cases, we start the game and lock the device to force system idling.

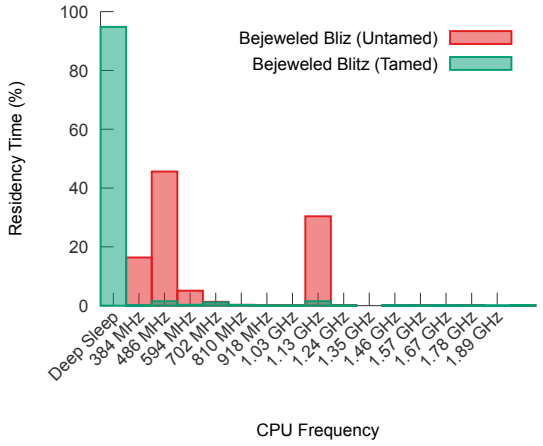


Figure 4.7: Relative CPU residency for the untamed and tamed versions of Bejeweled Blitz on the Galaxy S4.

`FullPower Recording`, and `NlpWakeLock`. Judging from the wakelock names, it is safe to assume that a few hardware subsystems remain awake while the application runs. We found an ebug after pausing a running session and locking the device screen. Although we expected the application to release all wakelocks while in the background, Nike+ Run Club kept them acquired. To counter this unwanted effect, we defined a policy similar to the case of Bejeweled Blitz: block the culprit wakelocks during background time. Figure 4.8 shows the battery drop for an eight-hour session before and after TAMER's interception. We see a 5× improvement in battery drain. Figure 4.9 presents the CPU residency for both scenarios: the deep-sleep residency jumped from 0% to 89.8%. We also acknowledge a major contribution of the GPS and other sensors to battery decay.

Their duty cycle is equivalent to the time the homonymous wakelocks were held.

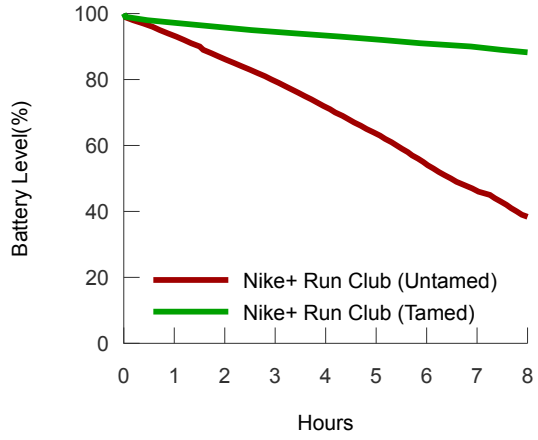


Figure 4.8: 8-hour battery drain on the Galaxy Nexus smartphone for Nike+ Run Club before and after being tamed. In both cases we start the app and lock the phone's screen to force idling.

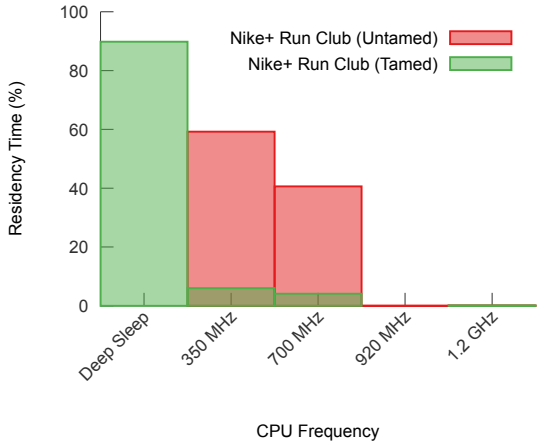


Figure 4.9: Relative CPU residency for the untamed and tamed versions of Nike+ Run Club on the Galaxy Nexus.

4.5.3 Performance Impact

Just like a network firewall, TAMER intercepts every hooked event call, inspects its signature, evaluates policy criteria, and finally actuates according to the policy's instructions. Although TAMER diverts the normal flow of applications, it should incur as little performance overhead and energy burden as possible. We instrumented

TAMER to measure the time taken to hijack an event and perform its blockage. For the longest diverted execution flow, TAMER required an extra 320μs (approximately 3,125 events/s) to execute on the Galaxy Nexus device when compared to a no-diversion code path. With regards to battery impact, we also measured the time taken to drain the battery of the device under test while running the same GMS background workload with and without TAMER. The time difference was negligible. TAMER is activated only when other applications generates background events and keeps its bookkeep data in memory for performance reasons. Furthermore, most of the time, it does not acquire any wakelocks but freeloads the system's active state from other wakeup sources, except when it needs to persist its logged data, which happens sporadically.

4.6 Challenges, Limitations, and Extensions

TAMER's main utility comes from policy definition, which is barely covered in this chapter. In the following, we expose our observations after experimenting with manual policies for TAMER and elaborate a research plan to automate the discovery of policies. We also suggest other improvements to TAMER that are left as future work.

4.6.1 On Policy Definition

In §4.5, we demonstrated how a wise policy selection can partially inhibit the surge of energy-hungry events. Our experience defining policies arose from intuition, reading source code (when available), and, in some cases, multiple trial-and-error attempts. Application developers and power users might not be inclined to follow the same route. Instead, they would benefit from the following guidelines to explore the event space and define effective policies. For a more streamlined solution, Chapters 5 and 6 detail our pursuit of tools that semi-automate the discovery of effective policies.

Choosing events to control. Handhelds include dozens of applications that generate hundreds of events. A policy maker should not consider all events as equally important. First, users prefer some apps over others [80]. Second, event triggers do not follow a uniform distribution. As a rule of thumb, developers should start with policies targeting the most frequent events.

Cutting the red wire. Even after selecting the most prominent events for policy testing, there are no guarantees that a policy will work without side effects. Side effects may include an increase in the frequency of correlated events, the rise of unexpected events, and abnormal application behavior. Blocking alarm events recklessly could, for instance, totally defeat the purpose of a calendar application. Because most apps are only available in binary form, understanding the purpose of an event is not always clear and neither is uncovering its dependencies. Techniques used in black-box testing, such as cause-effect graphs [170], can help. Events may also show a temporal correlation with others. In Chapter 5, we present a mechanism to uncover temporal dependencies between events.

4.6.2 Potential Improvements

Event batching over cancellation. TAMER dismisses event continuation if there is a need for throttling. An implementation that favors batching over cancellation would reschedule the asynchronous delivery of an event to coalesce multiple wakeups into one. AlarmScope [177] uses this approach to postpone the execution of non-critical periodic tasks. After observing the high impact of periodic background tasks on battery drain, Google decided to officially make all alarm events deferrable from the Lollipop version of the Android OS [87].

Task coalescing is actually a pervasive technique. For instance, the Linux tickless kernel [211] reduces the precision of software timers to allow for the synchronization of process wakeups, thereby minimizing the number of CPU power-state transitions. Xu et al.'s recent work on coalescing events to save energy in the context of e-mail synchronization [233] is another successful example of careful event handling in mobile devices. As long as developers do not assume guarantees on event delivery and commutativity, we believe coalescing should supersede cancellation as an energy-saving feature.

Native code support. TAMER controls apps by intercepting calls to functions of the Android's Java API. Applications that make heavy use of native code, such as games, multimedia apps, and ELF libraries, can acquire wakelocks, spawn threads, and perform background tasks using C/C++ code, thereby bypassing our control system. Extending support to native code would require a similar effort on analyzing and instrumenting `libc` function calls. Probing tools like SystemTap [30] and Cydia Substrate [203] can aid our instrumentation.

Support for other mobile OSes. Background processing is not exclusive to Android, although handled differently by other mobile OSes. Apple's iOS 7 and upper versions regard background processing as a privilege [43]. Other than network transfers, common background tasks have limited time to complete tasks and must respect the device's will to sleep. Background tasks are restrained to process data in chunks when the device wakes up to handle phone calls, notifications, or other interruptions. Windows Phone forces background tasks to be lightweight by applying quotas to resources like CPU, memory, and network when apps are running behind the scenes [164]. As a result, event-frequency control may not produce the same battery gains on Apple's and Microsoft's mobile devices given their stricter stance on deploying background tasks.

Feedback control. Tamer works as an open-loop controller, not using feedback data to gauge whether the system needs more adjustments. During the design stage of Tamer, we discarded the closed-loop approach as it would require knowledge of application semantics as well as user perception of performance degradation. Modeling these two elements are hard problems that are beyond the scope of this research.

Security issues. Xposed patches software code at runtime to modify its behavior. Because it has privilege access to inter-process communication and reserved system resources, there is a risk of exposing the system to attacks from malicious payloads. As a safer alternative, we could implement the monitoring and controlling modules of Tamer as part of the Android task scheduler. To keep compatibility, these extensions would need to be upstreamed to Android vendors for inclusion into their systems.

4.7 Related Work

We are not the first to propose control of functionality in exchange for battery savings. Tamer builds upon a number of contributions.

Android tasks killers once were the solution for background power savings, but their effectiveness is now a point of contention [33, 149]. Task killers force background applications to quit, assuming that their removal from memory will reduce the energy footprint of released resources. This is not always the case as there is little correlation between memory and CPU usage in Android [40]. Excessive task killing may lead to the opposite

effect – discarding cached data forces Android to reload apps from storage. Moreover, a killed app may restart itself immediately after being killed, raising CPU time and draining even more battery.

Rather than killing background tasks, popular battery-saving apps like JuiceDefender [142] and DU Battery Saver [92] can configure the access to power-greedy subsystems, such as the radio and the GPS, on a scheduled basis. Although effective in many cases, these solutions are coarse-grained as they focus on access to subsystems that are shared by many applications. Greenify [100] is an Android tool for hibernating apps, preventing the arrival and dispatch of events once there is a switch to the background. Original functionality is only restored when the blocked app returns to the foreground. Greenify is effective in blocking misbehaving and start-at-boot applications, but its treatment of background computing is coarse-grained and not applicable to notification-based apps that mostly run in the background (e.g., mail readers, instant messengers, calendars, etc.) Tamer is applicable to such cases as it throttles, but does not completely eliminate, background functionality.

The Android OS also includes its own controls for background-task management. Users can choose between enabling or disabling networked-data synchronization in the background. Until the release of Android Lollilop, this control had system-wide implications but later was turned application-specific. Tamer complements Android by letting users manage other types of background events not related to networking.

Partial inspiration for deep event monitoring stems from applications like BetterBatteryStats [138] and Wakelock Detector [223]. Both apps report wakelock-usage statistics that developers can use to understand the root cause of battery drain. Tamer complements these tools by empowering developers to take action after they pinpoint the origin of abnormal energy consumption.

Carat [174] and eStar [167] use collected data from thousands of smartphone and tablet users to profile the battery drain of applications and suggest alternatives. For instance, both tools suggest to kill or uninstall energy-inefficient apps. eStar further recommends energy-efficient alternatives to power-hog apps. Tamer let users keep their favorite apps while modifying the culprit's behavior to reduce its energy consumption.

4.8 Summary

This chapter presented TAMER, an OS mechanism that interposes on task wakeups in Android and allows event handling to be monitored, filtered, and rate-limited. We demonstrated that TAMER substantially reduces the background energy use of popular Android applications. With TAMER, a device spends more time in low-power mode, which increases the battery lifetime significantly.

While this chapter shows TAMER's effectiveness as a control mechanism, the following chapter focuses on defining policies that drive TAMER to reduce the energy burden of mobile applications. Future work is needed to determine how to select policies that not only reduce energy consumption but also minimizes the user-perceived functionality impact.

Chapter 5

Meerkat

Facilitating Policy Discovery to Drive Energy Control

Chapter 4 introduced a control mechanism to modify binary apps at runtime. In this chapter, we consider the process of defining Tamer policies to effectively lengthen battery lifetime.

5.1 Introduction

System-level energy management can be achieved via hardware or software adaptation. A number of hardware components provide the OS with configuration knobs that can be switched for the benefit of energy savings or performance gains. Software adaptation relies on changes to the running program. An application can opt for an energy-efficient code path over an energy-oblivious one [42]. A more drastic approach would skip portions of the running code [212] or delay and batch-schedule tasks to synchronize their execution with system wakeups [161,177]. Software adaptation can be introduced at compile time [47,208] or at runtime [120, 159,165].

Hardware- and software-adaptation solutions, despite their implementation differences, share an abstraction: a *control* mechanism that dictates how to modify the system. Adaptation is conditioned to external

factors, thereby requiring a means to communicate the circumstances that activate changes. For instance, let's consider the case of heterogeneous computing on handhelds. Modern handhelds employ heterogeneous CPU dies pairing small and efficient cores with larger and more complex cores to balance battery discharge against computing speed. An adaptive system will dynamically resort to the most appropriate core type according to the instantaneous performance requirement [8]. Here we have a threshold parameter that defines the boundary between high performance and low power draw.

Tamer is a software-control mechanism that alleviates the energy burden of background software tasks. Tamer also relies on parameters that specify when and where adaptation occurs. In both cases, *policies* embody a contract specifying the control parameters. Mobile devices include a myriad of other control-policy systems focusing on energy management and acting on the OS and applications levels (e.g., DVFS for CPU and GPU cores, screen timeouts, etc.) The effectiveness of these policies is highly dependent on the user context and there are often complex interdependencies that make it difficult to determine the optimal parameters for a given situation.

To alleviate this problem, we introduce Meerkat, a combined system that helps discover policy targets for Tamer to curb the energy demands of Android applications. Meerkat profiles the power draw of software events at function level, correlates and ranks the sequences of events based on their energy consumption on a multitasked system. Developers can use Meerkat's output to identify what events possibly consume the most energy and then write rate-limiting policies to reduce or even eliminate energy hotspots. We demonstrate Meerkat's effectiveness by using it in conjunction with Tamer's control mechanism.

We evaluate Meerkat against a specific but common mobile scenario: background battery drain. Reducing the execution frequency of background events can lead to substantial energy savings but requires expert knowledge on the system's internals and a microscopic view over the entangled streams of events coming from multiple sources.

The main value of Meerkat is the semi-automatic discovery of the root causes of excessive energy consumption inside applications. More importantly, because Meerkat tracks causal relations between events, developers can quickly identify and cut the nip in the energy-bud without spending hours debugging code.

5.2 Motivation

Previous works have demonstrated that many Android apps suffer from serious energy-inefficiency problems [153,181]. Locating these problems is labor-intensive and automated diagnosis is highly desirable. Developers have to extensively test their apps on different devices and perform detailed energy profiling to identify the root causes of energy problems and opportunities for optimization. Our initial experience with Tamer was not different. Tamer controls the execution of event abstractions via policies. Our main challenge is to find what events spend the most energy so that we can write controller payloads and policies that make sense.

Figure 5.1 graphically demonstrates that the Android OS can dispatch a multitude of events almost simultaneously. Finding the energy contribution of each event is a daunting task. First, reading the battery state via direct OS polling or from external measurement hardware only discloses the power draw of the entire system. Power models, on the other hand, attempt to characterize the energy portion of each subsystem and attribute the sum of energy parts to fine-grained software entities based on how these entities use each subsystem. As previously demonstrated [63,162], this approach is prone to inaccuracy. In the case of reactive events, a common pattern in smartphone applications, estimation errors may compound as some events may only run for a few microseconds, which can lead to overestimated power models.

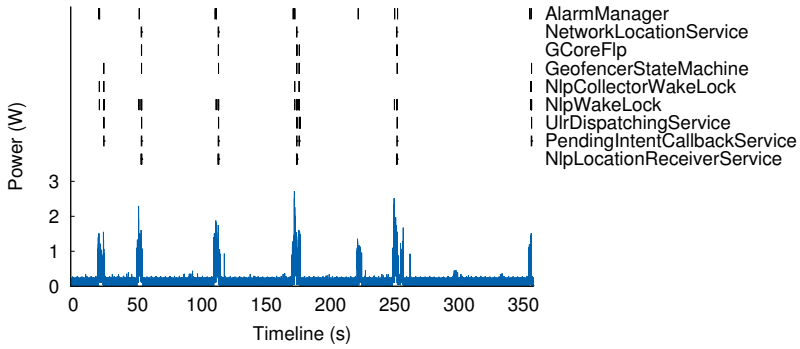


Figure 5.1: Partial reproduction of Figure 4.2. Events arising from different applications tend to agglutinate when running in the background.

In conclusion, individual energy apportioning is challenging because the available context information

is insufficient to properly allocate Joules to individual function calls. Multitasking, task parallelism, asynchronicity, and hardware-resource sharing only exacerbate the problem. As well stated by Dong et al. [90], "it is practically infeasible to track how software uses each hardware component in a heterogeneous multicore system like modern mobile devices."

Being so, we take a simpler approach to pinpoint energy-hungry events: instead of precisely attributing energy consumption to singular function calls, we focus on finding sequences of event calls that are *likely related to high energy use*. These call sequences will serve as input for effective Tamer policies. To do so, we rely on a time-analysis technique named *sequential pattern mining*.

5.3 Design and Implementation

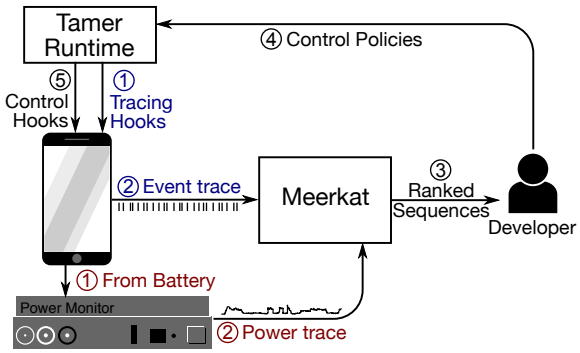


Figure 5.2: Meerkat's task pipeline.

Meerkat extends Tamer's instrumenter-and-monitor pipeline by adding a data-mining engine. Figure 5.2 depicts our task pipeline. The instrumenter is responsible for installing hooks on selected framework and library code. We use hooks to log the execution of particular events and rate-limit their execution at runtime, if necessary. The data-mining engine is used to correlate the dynamically traced events with the power draw of the handheld's battery. In the following, we elaborate on the subtasks necessary to realize our system.

5.3.1 Data Collection

Acquiring Power Traces

We use a profiling computer to drive the execution of tasks on the handheld. To do so, we connect the profiling computer to the handheld via a USB cable and use the `adb` (Android Debug Bridge) program to send commands that start, control, and stop mobile apps. To obtain power traces, we connect the Monsoon power monitor to the handheld's battery. The profiling computer instructs the Monsoon power monitor on when to start and stop collecting power samples at a frequency of 1KHz, and store these samples as they are captured.

Obtaining Event Logs

We use Tamer's monitor module to obtain more detailed execution logs. Tamer intercepts a narrow yet well-defined interface of functions that Android app developers use to dispatch background tasks. This interface consists of `Services` and `BroadcastReceivers`, which are scheduled once or periodically using an `Alarm`. We extended the set of captured background events, thereby increasing code coverage. Although `Services` and `BroadcastReceivers` are the entry points to background work on Android, other asynchronous methods can be dispatched once the entry door is crossed. Tamer's updated monitor also instruments `Threads`, `Runnables`, `Handlers`, `AsyncTasks`, and `Executors`. Table 5.1 shows the updated list of hooked functions.

Some of these event-calls are so pervasive in system code that their interception provides almost no useful information yet capturing them adds much overhead. Tamer keeps a blacklist of such calls to balance code coverage against overhead. This blacklist contains a handful of `Thread` and `Executor` class signatures involved with on-screen and off-screen drawing.

Each collected log spans 12 hours of running time, starting from a full battery to avoid different system behavior when the device is running out of battery. At the end of a test run, we download the event logs from the handheld via `adb` for offline processing.

Event Type	Event Sub-Type	Parent Class	Hook Point (Method)
Wakelock	-	`com.android.server.power.PowerManagerService`	`acquireWakeLockInternal`
Alarm	-	`com.android.server.AlarmManagerService`	`triggerAlarmsLocked`

Service	Service
	IntentService
	JobService
		`com.android.server.am.ActiveServices`
		`android.app.Service`
		`android.app.job.JobService`
			`startServiceLocked`
			`onHandleIntent`
			`onStartJob`

BroadcastReceiver	-	`android.content.BroadcastReceiver`	`onReceive`
Thread	Thread TimerTask	`java.lang.Thread` `java.util.Timer.TimerImpl`	`resume` `run`
AsynTask	-	`android.os.AsyncTask`	`doInBackground`
Handler	-	`android.os.Handler`	`handleMessage`

Table 5.1: Updated instrumentation points of TAMER.

Synchronizing Data Inputs

To correlate application events with power peaks, we must first align the power traces with the event logs. This is a necessary step since the power and event sources have distinct clocks and there is a potential for drifting.

There are several ways to achieve synchronization. For instance, NEAT [63], a power-monitoring and software-analysis solution, uses an invasive approach to synchronize different signals. Brouwers et al. replaced the phone's vibrating alert motor with a wire to trigger the power meter. BattOr [207], on the other hand, modulates the smartphone's LED camera flash as a pseudo-random recognizable pattern to align the measured current with the device's execution timestamps.

In a similar way, Meerkat uses the power draw itself as a means to synchronize different signals. Assuming a handheld idling with the screen off, the profiling computer instructs the power monitor to start and stop collecting current and voltage samples. At the same time, the profiling computer generates a pulse in the power trace by turning the display on and off at the start and end of an experiment. To do so, we connect the profiler to the mobile device using Monsoon as a bridge.

The Monsoon PM has a USB passthrough mode that can be switched on or off. The passthrough mode establishes a USB data connection between the handheld and the profiling machine, which allows for downloading files, configuring test programs, and charging the device [168]. Once we start the power sampling, the USB connection is disabled, and samples are measured without interference of charging. As a side effect of disabling USB charging, the Android OS turns on the screen of the idling device. By programmatically configuring the screen timeout to a few seconds, we generate recognizable pulses on the power trace.

As the display is turned on and off, Meerkat logs two events onto the execution log to indicate the timestamps of the recognizable signal: `android.intent.action.SCREEN_ON` and `android.intent.action.SCREEN_OFF`. Using these events, we are able to align and cut the power and execution timelines. We also time-stretch one of the timelines relative to the other to eliminate any potential clock drift.

5.3.2 Sequence Event Mining

The analysis of relationships between variables is a fundamental task at the heart of many data-mining problems. There is a variety of methods to mine temporal patterns from sequence datasets, such as mining repetitive patterns, trends, and sequential patterns (see [195] for a survey of methods). Among them, sequential pattern mining is probably the most popular set of techniques [37] and consists of finding "interesting" subsequences in a set of sequences. Interestingness is a suitable measure to evaluate the dependencies between variables and takes various forms. Support (frequency), correlation, interest factor, and entropy are some examples [219], with support being the most commonly used.

Knowing that a sequence appears frequently is not sufficient for making predictions [108]. To avoid dispatching energy-hungry events, Meerkat should predict which function is called next in a sequence to prevently block it. An alternative that addresses the problem of prediction is sequential rule mining [85,155,160]. A sequential rule indicates that if some event(s) occurred, some other event(s) are likely to occur afterward with a given confidence. Compared with sequential patterns, sequential rules can help users better understand the chronological order of sequences present in a sequence dataset.

Sequential rule mining has been applied to many applications areas, including stock management [57,62], DNA sequence analysis [145], recommender systems [131, 132], and software engineering [154, 235]. In this chapter, we adapt these pattern- and rule-mining techniques to the problem of energy analysis. We first identify energy-relevant sequential patterns and then derive sequential rules from these patterns.

A sequence is an ordered list of events. A sequence pattern is a sequence or subsequence recognized from a dataset. Finally, a sequence rule is a rule of the form $X \rightarrow Y$ where X and Y are event-sets. An event-set consists of one or more items (events) ordered by time. For the sake of simplicity, we assume that no events

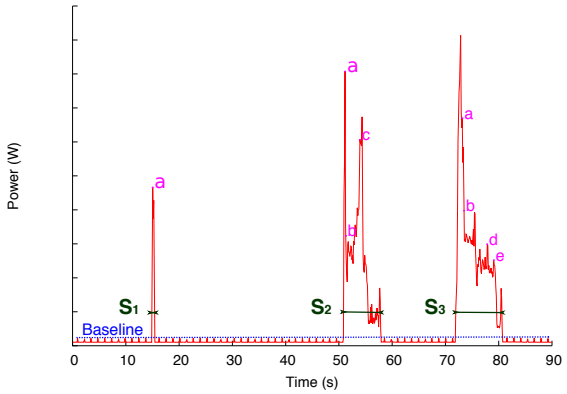


Figure 5.3: Power intervals associated with event sequences.

occur simultaneously. An input event is presented as a triplet (`eid`, `time`, `value`), where `time` is the time when event `eid` occurs and `value` is a set of attribute values associated with the event. In our case, `eid` is the full name of the function captured by TAMER in the execution log (i.e., the function name prefixed by its package and class owner) and `value` contains the application name owning `eid`.

Figure 5.3 shows a snapshot of an event dataset adapted to sequence mining. We define a sequence interval as a contiguous time window in which the measured power is significantly above the idle baseline. A sequence corresponds to a subset of events contained inside a power interval. Figure 5.3 includes three sequences: S_1, S_2, and S_3. Each event belonging to a sequence is identified by a lowercase letter. Taking S_2 as a dataset example, the possibly valid sequential patterns generated via extension are: $\langle a \rangle$; $\langle b \rangle$; $\langle c \rangle$; $\langle a, b \rangle$; $\langle a, c \rangle$; $\langle b, c \rangle$; $\langle a, b, c \rangle$. From these sequential patterns, we can derive the following rules (we omit rules starting with or ending in an empty set, since they are not useful for our purpose): $\langle a \rangle \rightarrow \langle b \rangle$; $\langle a \rangle \rightarrow \langle c \rangle$; $\langle b \rangle \rightarrow \langle c \rangle$; $\langle a \rangle \rightarrow \langle b, c \rangle$; and $\langle a, b \rangle \rightarrow \langle c \rangle$.

The classical approach to mine sequential patterns uses *support* as an interestingness metric. Given a dataset containing various sequences, the support of a pattern $X \cdot Y$ is the number of sequences that contains the events in X followed by the events in Y divided by the number of unique sequences in the dataset – $\sup(X \cdot Y)$, where $\cdot$ indicates a concatenation of event-sets. The pattern miner outputs all the patterns whose support are above a given threshold *minsup*. From this output, the rule miner uses another interestingness metric, *confidence*, to rank the most frequent rules. Given a pattern $X \cdot Y$, the confidence of the derived rule

$X \rightarrow Y$ is $\sup(X \cdot Y)$ divided by $\sup(X)$, i.e., $P(X|Y)$.

When hunting for energy-inefficiency in software, solely counting function dispatches is not sufficient, as rare yet energy-hungry events can also occur. To find energy-relevant sequential patterns, we redefine the support metric – instead of counting the number of event calls, we base the support of a subsequence S_i on the energy intake of all power windows that contain S_i.

There are many approaches to allocate energy intake to subsequences or individual events running in parallel, with equal split, differential apportioning, and weighted distribution being the most prominent examples. Yet, any strategy is an approximation of the ground truth we cannot obtain, since we use the system's power draw as our source of energy knowledge.

We pose that, in the lack of better provenance data, every event or subsequence belonging to a power window, regardless of task parallelism, should be held equally responsible for the energy consumption of the power window to which it belongs. For example, if S_2 consumes 3J, all of its event members ($\langle a \rangle$, $\langle b \rangle$, and $\langle c \rangle$) and extended subsequences ($\langle a, b \rangle$; $\langle a, c \rangle$; $\langle b, c \rangle$; and $\langle a, b, c \rangle$) should receive the same weight of 3J.

We define the new energy-support of a pattern $X \cdot Y$ as follows. For a sequence S_i that satisfies $X \cdot Y$, its energy-support is the sum of the energy in the power windows containing S_i divided by the sum of the energy in all power windows.

Searching for Sequences

The task of sequential pattern mining is an enumeration problem. It aims at enumerating all patterns (subsequences) that have a support no less than the minimum support threshold (*minsup*) set by the developer. Discovering sequential patterns is a hard problem. The naïve approach calculates the support of all possible subsequences in a sequence dataset and then outputs only those meeting the minimum support constraint. Such a naïve approach is inefficient because the number of subsequences can be very large. A sequence containing n events can have up to $2^n - 1$ distinct subsequences, which makes the mining problem unrealistic for most real-life sequence datasets.

There are many optimized algorithms for discovering sequential patterns. Some of the most popular are

GSP [218], SPADE [239], PrefixSpan [182], SPAM [46], CM-Spam [107], and CM-Spade [107]. All of these algorithms explore the search space of sequential patterns by performing an *extension* operation. An extension operation generates a $(k+1)$-sequence (a sequence containing $k+1$ events) from a k-sequence. A sequence $S_a = \langle a_1, a_2, \ldots, a_n \rangle$ is a *prefix* of a sequence $S_b = \langle b_1, b_2, \ldots, b_m \rangle$, if $n < m$ and $a_1 = b_1, a_2 = b_3, \ldots, a_n = b_{m-1}$. For example, the sequence $\langle a, b \rangle$ is a prefix of $\langle a, b, c \rangle$.

In general, sequential mining algorithms can be categorized as either depth-first search (SPADE, PrefixSpan, SPAM, CM-Spam, CM-Spade) or breadth-first search (GSP). They start from the sequences containing single items (e.g., $\langle a \rangle$, $\langle b \rangle$, and $\langle c \rangle$), calculates their support, and proceeds to the next extension level if the calculated support is above *minsup*.

Since the search space of all possible subsequences in a dataset can be very large, designing an efficient algorithm for sequential pattern mining requires integrating techniques that avoid exploring the whole search space. The basic mechanism for pruning the search space in sequential pattern mining is called the *Apriori property*, also known as *downward-closure property* or *anti-monotonicity property* [36]. This property states that for any two sequences S_a and S_b, if S_a is a prefix of S_b ($S_a \sqsubset S_b$), then S_b must have a support that is lower or equal to the support of S_a. In the case of energy-support, it is easy to prove that this property holds because the number of power windows that contain all elements of S_b is always lower or equal to the number of power windows that contains all elements of S_a, thus energy-support$(S_b) \leq$ energy-support(S_a). The above property is useful for pruning the search space. As we show in the next subsection, we add a few exceptions to this procedure.

Dealing with Overestimation

A caveat of our weighting strategy is the possible overestimation of the energy-support, especially for smaller patterns. For instance, in Figure 5.3, the pattern $\langle a, b \rangle$ has a higher energy-support than $\langle a, b, c \rangle$ and $\langle a, b, d \rangle$. Being so, an energy-aware developer would write policies targeting the shorter pattern $\langle a, b \rangle$. In fact, blocking $\langle a \rangle$ as soon as it appears will result in the greatest energy savings, assuming that all of the other events depend on $\langle a \rangle$ being called. From the perspective of functionality, this solution is not desirable as it prevents all events

from running. In addition, developers might want to allow certain subsequences of events to proceed, while blocking others. For instance, a developer could block $\langle a, b, c \rangle$ while permitting $\langle a, b, d \rangle$ to run. Being too greedy prevents making choices based on functionality. If $\langle a, b \rangle$ is blocked, then neither one of the two longer subsequences will have a chance to run.

Concerned with these issues, we decided to filter out certain patterns as they are generated, following these considerations:

- If S_a is a prefix of S_b and energy-support(S_a) = energy-support(S_b), consider S_b as a target candidate for developer inspection, instead of S_a. Thus, we add an exception to the anti-monotonicity property: to not prune the search space of a subsequence if it is a prefix of a longer subsequence with the same energy-support.

- Only admit sequential patterns that are longer than three events, as long as said pattern appears in a single power window and belong to the same application. We decided on the number three after observing the CDF on the number of events in a sequence, belonging to the same application, and restricted to the same power window (Figure 5.4). Sequences with three events are between the 32^{th} and 78^{th} percentile of all sequences we have observed from our learning data, thus being the most common. It is important to note that this distribution directly depends on the code coverage of Tamer hooks. Adding or removing hooks affects the distribution of length of discovered sequential patterns. Figure 5.4 already considers the additional hooks we will define in Chapter 6.

The output of the pattern miner is a list of patterns sorted in descending order by their energy-support. We use the pattern length as a secondary sorting parameter, so that longer sequences that have the same energy-support as smaller ones appear first in the ranking.

Generating Relevant Energy Rules

The sorted output of sequential patterns serves as the starting point for a developer to examine code as well as to derive runtime policies that rate-limit the execution of energy-expensive sequences of events. First we

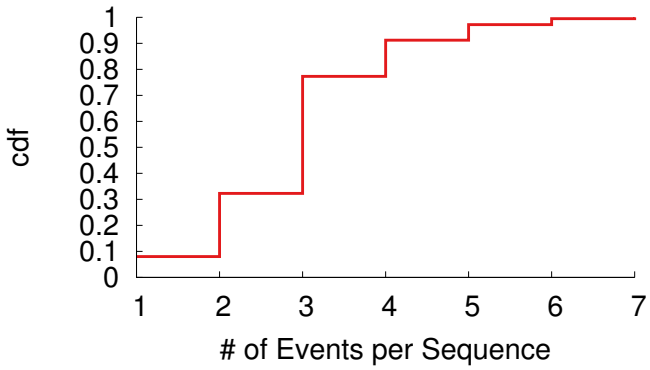


Figure 5.4: CDF of the number of events in a sequence belonging to the same app.

generate all possible rules from a given pattern. For instance, from the pattern $\langle a, b, c, d\rangle$, we can generate three different rules: $\langle a\rangle \rightarrow \langle b, c, d\rangle$; $\langle a, b\rangle \rightarrow \langle c, d\rangle$; $\langle a, b, c\rangle \rightarrow \langle d\rangle$. We use the confidence metric as a hint for developers to decide at which moment of a sequence call a block should occur. We pick the rule with the highest confidence to decide at which event we should block a sequence. For instance, if $\langle a, b\rangle \rightarrow \langle c, d\rangle$ is the rule with the highest confidence, we should set up a policy that blocks event c in when $\langle a, b\rangle$ has preceded it.

5.3.3 Extending the Controller

The sequential rules mined from the power and event data indicate the possible energy-hungry events. An important distinction from our previous iteration of Tamer is that we now must assess the energy consumption of applications when events occur in conjunction. Therefore, in order to transform mined rules into policies, we must extend Meerkat's runtime controller to match sequences of events. Each suggested rule, after filtering, becomes a policy target that is tested at runtime for energy savings. The policy target corresponds to the sequential pattern in the mined rule.

We implemented a stream-oriented version of the Knuth-Morris-Pratt algorithm [139] (KMP) to match the running events with the policy target. The original KMP algorithm searches for occurrences of a "pattern" P within a main "text string" T by employing the observation that when a mismatch occurs, the pattern itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. This optimization leads to a running time of $O(n + m)$, where n is the size

of T and m is the size of P, which is optimal in the worst-case sense. The KMP algorithm preprocesses the pattern P by computing a failure function f that indicates the possible shift s, necessary to locate how far the observed input is from pattern P, using previously performed comparisons. Specifically, the failure function $f(j)$ returns the length of the longest prefix of P that is a suffix of $P[i \ldots j]$. Pre-computing this function requires $O(m)$ extra space. For each installed target policy, we must compute a new function table.

Our modifications are trivial: (1) Our pattern P is the policy target (an ordered sequence of events). Each element of P is an event signature; (2) Instead of using character matching, we use hash search to compare each element of P with the runtime-monitored event; (3) Instead of using a static corpus T, we use a rolling window representing the monitored events as they are executed. T can be streamed in because the KMP algorithm does not backtrack on the input.

Meerkat only blocks an event if it is the last element of a pattern P and there is a full match between P and the runtime input. Just like Tamer, blocking depends on the rate defined in the controller policy. While Meerkat identifies possible event targets, the policy designer is responsible for defining rate limits.

5.4 Preliminary Evaluation

We conducted experiments on a Samsung Galaxy S4 device (cf. Table 5.2) running AOSP KitKat. We evaluate Meerkat in two ways. First, we revisit the GMS-evaluation scenario in §4.5 to demonstrate that Meerkat can uncover energy-relevant rules that lead to policies as effectively as the ones we manually selected in §4.5. We then elaborate on Meerkat's performance and energy overhead. Chapter 6 discusses specialized extensions of Meerkat to particular app domains and demonstrates the effectiveness of Meerkat on a variety of scenarios.

Application Processor	Quad-core 1.9GHz Qualcomm Krait 300
Communication	4G LTE radio on
Localization	GPS on
OS Version	AOSP KitKat (4.4)

Table 5.2: Specification of device under test (Samsung Galaxy S4 smartphone).

5.4.1 Revisiting Google Mobile Services

Automated energy diagnosis is highly desirable. Figure 5.1 shows the frequent power peaks associated with the GMS-based installation on AOSP KitKat. Along with the power trace, there are entangled streams of events arising from multiple sources. Finding the energy contribution of each event is a daunting task. Nonetheless, we used this representative scenario as part of Tamer's evaluation. To obtain our final selection of policy targets in §4.5.1, we had to manually verify multiple combinations of policies. In some of our trials, the lack of a policy target would prevent an increase in battery lifetime, whereas in other cases the inclusion of a potential target would keep the lifetime the same.

To test Meerkat's effectiveness, we reproduced the GMS scenario of §4.5.1 by first collecting power traces and event logs while running the Android OS in the background with the GMS-base install. Next, we used Meerkat's mining engine to obtain a ranking of the potential event policy targets. Finally, we generate new Tamer policies using the same rates proposed in §4.5.1 and compare the lifetime gains with the ones obtained manually.

Table 5.3 shows the relevant rules unveiled by Meerkat. In total, we found four rules that potentially cover the most energy-relevant event combinations related to GMS. Three of these rules proved sufficient, when applied together, to obtain similar battery improvements as in §4.5.1.

Rule	Ranking Position	Effective?
$a, b \rightarrow c, d$	2	Y
$e \rightarrow f, g$	5	Y
$a, f \rightarrow c, d$	8	Y
$a, i \rightarrow c, j$	14	N

Table 5.3: Relevant sequential rules extracted from the GMS-based running scenario. For better readability, event names are represented by letter symbols. Table 5.4 serves as a translation guide.

5.4.2 Performance Impact

Given that data collection and correlation take place on the profiling computer, we refrain from accounting Meerkat's performance impact on the device under test. To assess the performance impact due to the updated

Symbol	Event Name (Class.Method)	Event Type
a	com.google.android.gms.nlp.ALARM_WAKEUP_LOCATOR	Alarm
b	com.google.android.location.network.NetworkLocationService.onStartCommand	Service
c	NlpWakelock	Wakelock
d	android.os.HandlerThread.run	Handler
e	com.google.android.gms.nlp.ALARM_WAKEUP_BURST_COLLECTOR	Alarm
f	com.google.android.location.fused.NlpLocationReceiverService.onStartCommand	Service
g	NlpCollectorWakelock	Wakelock
h	com.android.server.LocationManagerService	Wakelock
i	com.google.android.location.activity.HardwareActivityRecognitionProviderService.onStartCommand	Service
j	com.google.android.location.internal.PendingIntentCallbackService.onStartCommand	Service

Table 5.4: Translation table of symbols to events (cf. Table 5.3).

version of Tamer, we measured the time taken to hijack, log, and block an event, just as we did in §4.5.3. We discovered that, for the longest diverted execution flow, Tamer takes, on average, 380μs (approximately 2,630 events/s) to execute on the Galaxy S4 device. We also measured the time taken to drain the battery of the device under test while running the same GMS background workload with and without Tamer. The time difference was negligible, since Tamer only activates itself when an application generates a hooked background event.

5.5 Discussion

Meerkat has a series of limitations, including:

Impact of the Observer Effect. The observer effect refers to changes that the act of observation reflects on a phenomenon being observed. This is often the result of instruments that, by necessity, alter the state of what they measure in some manner. There are at least two potential sources of changes introduced by Meerkat. First, by wiring the Monsoon power monitor to the device's battery, we affect the current that the power monitor measures by way of presenting an additional load to the battery circuit. Second, by instrumenting the source code with additional hook points we temporarily divert the code-path execution to Tamer's controller payload. Although we show in §5.4.2 that the computational overhead of logging and blocking events is minimal, it does not guarantee that the system behavior will remain intact. Finally, we resorted to artificial measures to reduce variability in our tests. These include not setting up certain software known for their unpredictable behavior while running (e.g., apps often receiving network packets at random times due to external factors) and using a more stable network connection (WiFi instead of cellular connectivity).

Lack of Coverage of Important Events. Finding statistically relevant patterns relies on reporting the relevant data. We treat power traces as the ground truth and collect them at a resolution high enough to cover power shifts down to the order of microseconds. Events logs, on the other hand, have a series of shortcomings that can affect Meerkat's coverage and, consequently, its effectiveness.

- Meerkat's instrumentation covers a well-defined interface used for dispatching background tasks. Yet, it does not account for all of the power draw associated with an interval of high activity. Meerkat lacks cross-layer tracking. By design choice, we opt for monitoring only application code running inside the Android sandbox. Yet, an executing task will eventually cross the userspace boundary and reach the kernel space and finally the device drivers. The energy consumed by the latter two layers is not properly apportioned but distributed to all tracked Android events belonging to the same power interval.

- The use of anonymous inner classes in Android code convolutes the identification and separation of events by Meerkat. An anonymous inner class is a class not given a name and is both declared and instantiated in a single statement. Anonymous classes are created for a specific purpose inside another function (e.g., as a listener, as a runnable (to spawn a thread), etc.) and always implement an interface or extend an abstract class. Tamer mostly monitors function calls coming from subclasses of an interface or abstract class used to dispatch background tasks (cf. Table 5.1). Because these classes are nameless, Meerkat is not able to differ their signature. As a result, the data-mining engine attributes all of the energy spent by anonymous classes to a single bin, thus leading to overestimated energy-support apportioned to sequences that include a function call belonging to an anonymous inner class. To avoid this unwanted behavior, we opt to disregard activities occurring on behalf of anonymous classes in exchange for less code coverage.

- Event dispatching coming from the Reflection API in Java, instead of direct procedure calls, is another source of convolution that reduces Meerkat's accuracy in identifying events. Reflection can be used to obfuscate code against decompilation. Reflection is also key in frameworks that promote *inversion of control* [109], a software-engineering technique used to promote code modularity and make it extensible.

RoboGuice [25], Dagger [128], and ButterKnife [229] are inversion-control frameworks for Android that enjoy great popularity among Android developers, especially in popular applications [1].

5.6 Related Work

On Policy Inference. EASEAndroid [227] is an automatic platform for analysis and refinement of security policies for the Android OS. It uses semi-supervised learning and continuous feeding of audit logs to harden the OS security. AutoPPG [236] leverages static code analysis to construct correct and readable descriptions of an app's behavior related to personal data. These descriptions facilitate the generation of privacy policies for Android apps. Similarly, Meerkat acts as an intermediate step that facilitates the generation of control-execution policies aiming at energy efficiency for Android apps.

On Code Analysis. There is a variety of frameworks designed for the analysis of Android applications. These frameworks rely on static [59, 83, 98, 113] or dynamic analysis [59, 64, 73, 74, 118, 122, 158] and are used for different purposes: inspect security properties, automate GUI tests, or identify energy hotspots.

Schmidt et al. [204] used machine learning techniques to identify malicious binaries in Android applications by extracting and classifying function calls found in ELF binaries. RiskRanker [113] and AnDarwin [83] are scalable frameworks that provide valuable insight into an application's behavior. RiskRanker uses a plethora of static-analysis techniques including program-control-flow graph-evaluation and assessment of bytecode signatures. AnDarwin uses program-dependency graphs to detect plagiarized applications. RiskRanker and AnDarwin are relevant because they address the issue of scale. Nonetheless, Meerkat requires insight into runtime behavior, which is why we opt for dynamic analysis in our implementation.

Crowdroid [64] crowdsources the dynamic analysis of Android applications and partially addresses one of Meerkat's limitation – scalability. An open question is how to identify and avoid skew in collected data due to different usage patterns.

Mahmood et al. developed a scalable dynamic analysis framework for analyzing Android apps in the cloud [158]. Their platform uses the Robotium test-automation framework [26] to drive the user interface

of applications. Robotium requires the application under test to be signed in debug mode. Because production applications rarely contain a debug signature, they must be re-signed to work with Robotium. As a result, applications may break or significantly reduce functionality if Android detects a non-official re-signature. Because we are interested in the dynamic energy analysis of Android apps, Meerkat aims to avoid conflicting behaviors that occur when an application is modified (luckily, modifications due to Xposed are not detected by the security mechanisms of Android).

ADEL [242] uses dynamic-taint analysis to identify energy leaks due to network communication. In ADEL's context, an energy leak is the use of energy by activities that never directly or indirectly influence the user-observable output of the smartphone. ADEL tracks the information flow of network traffic through applications and help developers isolate the causes of energy leaks by exposing the arrival time and contents of unused packets. Meerkat is complementary to ADEL as it works on a larger domain of energy leaks. In Chapter 6, we present an application of Meerkat in the network domain.

GreenDroid [153] is an automated approach to diagnose energy problems in Android applications. Just like Meerkat, GreenDroid uses dynamic analysis to explore the state space of an Android application. GreenDroid analyzes how sensor data are used by applications and monitors whether sensors and wakelocks are properly unregistered/released. GreenDroid stops short of proposing modifications to apps but elucidates where sensor-related energy hotspots reside inside an application.

On Activity Data Mining. MobileMiner [115] and ACE [171] are general-purpose services that run on smartphones and discover co-occurrence patterns indicating context events that frequently occur together. Commonly occurring patterns can be used to uncover and to express common-sense knowledge about user context. Patterns can also improve the user experience during the cold-start period when personal behavior is yet unknown. A series of works use specialized predictive classifiers and other targeted approaches [176,209,234,245] for phone optimizations, such as app launching and app preloading. These tools share with Meerkat the same principles, despite differing in their objective.

Gupta et al. presented a methodology for collecting, analyzing, and detecting energy anomalies from synchronized power traces and event logs [115]. Their approach differs from Meerkat's in that they use decision

trees to model the influence of whole software modules (i.e., libraries) on the average energy consumption of applications. Meerkat, on the other hand, uses a finer-grained approach, apportioning energy influence to bundles of time-related functions.

5.7 Summary

This chapter presented Meerkat, an energy-pattern analyzer tool for the Android OS platform. Meerkat associates power peaks with synchronized execution logs and help developers discover potential sources of energy hotspots. There are many ways that a developer can influence the power consumption of a mobile app. Developers can use Meerkat to find targets for Tamer policies from a series of event streams generated by running workloads on a mobile device. A major shortcoming in Meerkat is the possible lack of coverage for certain energy-relevant events, which we partially address in the next chapter.

Chapter 6

Meerkat in Action

Two Case Studies

6.1 Introduction

Meerkat relies on program analysis to determine code flow and track energy-related information during execution. To keep the system lightweight, Meerkat traces and acts upon a small set of asynchronous operation patterns provided to developers by the Android SDK. While our approach is effective in identifying prominent sequences of energy-hungry operations, it is also, in some occasions, prone to overestimation and not able to uniquely identify and correctly apportion energy to certain sequences of operations. As a result, developers miss the opportunity to discover valuable policy targets. This liability is a result of our design choice: we only trace commonly used patterns, which, sometimes, cannot be uniquely labeled (e.g., anonymous `Threads`) or often appear associated with many different subsequences.

We could naturally remove this limitation by adding extra tracepoints to compose sequences that uniquely stand out during execution. By increasing the coverage of a tracer, longer and unique event sequences can be logged and turned into policy targets. To guarantee uniqueness, this approach could be extended to the

point of covering every energy-intensive function call of all applications running on a device. However, our experience with Application Modes (Chapter 3) showed that point solutions do not scale. Therefore, we must reach a compromise between OS-generic calls and app-specific events.

In this chapter we demonstrate that such midpoint exists if, in addition to the tracepoints covered by Meerkat, we additionally trace events stemming from libraries linked by mobile apps. Libraries perfectly fit our purpose as they supply a well-defined interface to invoke behavior and they are reused by multiple independent programs. When a program invokes a library, it gains the behavior implemented inside that library without having to implement the behavior itself. Furthermore, a library's public interface tends to quickly stabilize; once we establish a tracepoint to one of its publics functions, it is unlikely that we will need to modify the tracepoint in the future even if an application's internal behavior changes.

We attest the efficacy of our approach by showing the improvements in battery lifetime from policies derived from logs captured using extended versions of Tamer: one covering event calls originating from Android network libraries; the other covering events from storage libraries. We base our case studies on some of the recent findings by Li et al. [146] in their empirical study on the energy consumption of over 400 Android applications as well as Lee and Won's study on I/O patterns of Android-based devices [144]. In summary, the authors discovered that:

- The total energy consumed by apps is dominated by the energy consumed by system APIs. This finding is harmonic with the design of Meerkat– we target API function calls for correlation with power peaks.
- The network interface is the most energy-consuming component in Android apps when the screen is off. In particular, making an HTTP request is the most energy-consuming network operation. NetDiet (§6.2) is an application of Meerkat's principles to detect and attenuate the background energy consumption of HTTP requests from Android and third-party network APIs.
- In Android-based smartphones, synchronous writes constitute a significant portion of all write I/O operations. Many smartphone apps use SQLite to manage persistent data, exploiting `fsync()` to preserve the atomicity of database operations. Synchronous operations reduce the chances of I/O batching, thus

consuming more energy. IODiet (§6.3) is an application of Meerkat's principles to detect and attenuate the energy consumption of database I/O requests from Android and third-party storage APIs.

6.2 Case Study 1: NetDiet

6.2.1 Design and Implementation

Whether an application retrieves data from a server, updates social-media status, or downloads remote files to disk, it is the HTTP network requests living at the heart of mobile applications that make the magic happen. Consider a developer implementing a weather widget on Android. A conventional weather widget fetches the city temperature from a remote server using a REST API. In the typical development cycle for Android, a widget developer would first write a `Service` code-block to request the weather data from a remote server. To handle the server response, the developer would also write a `BroadcastReceiver` to parse the received data and update the UI widget. In both cases, the developer must rely on network-programming classes to open a communication channel and to exchange data. This common pattern is found in many mobile apps.

In this section, we focus exclusively on data communication over HTTP and elaborate on the design and implementation of NetDiet. Since network exchanges can consume considerable energy, we first investigated how Android apps commonly access the network. We discovered that although many apps use Java native network classes (e.g., `java.net.URLConnection`), others rely on first-party and/or third-party network libraries that encapsulate the complicated and error-prone nature of network programming. We used libtastic [2] and AppBrain [1], two developer-oriented app-indexing websites, to find the most popular Android network libraries used in the top apps from the Android US marketplace. These are OkHttp, Retrofit, Apache HttpComponents and Volley (Table 6.1 lists the popularity of each library among developers). We studied the source code of each library, identified the hook points where network calls are dispatched, and wrote callbacks to trace execution and possibly block dispatches (see Table 6.2 for a list of all hook points). Interestingly, we discovered that these libraries build on top of each other (Figure 6.1 shows their relationship). Consequently, a call to a hooked function belonging to an upper library can lead to a longer, unique sequence trace.

Library Name	Popularity (AppBrain)	Popularity (libtastic.com)
OkHttp	18.91%	2nd
Retrofit	15.49%	4th
Apache HttpComponents	8.65%	1st
Volley	-	3rd

Table 6.1: Popularity of network libraries used in top-ranked apps. Sources: AppBrain [1] and libtastic.com [2].

Library	Class	Hook Point (Method)	Type
OkHttp	`com.squareup.okhttp.Call` `com.squareup.okhttp.Callback`	`execute` `enqueue`	Synchronous Asynchronous
OkHttp3	`okhttp3.RealCall`	`execute` `enqueue`	Synchronous Asynchronous
Volley	`com.android.volley.NetworkDispatcher` `com.android.volley.RequestQueue`	`performRequest` `add`	Synchronous Asynchronous

Retrofit	`retrofit.RestAdapter$RestHandler`
	`retrofit.CallbackRunnable`
	`retrofit.client.Request`
		`invokeRequest`
		`run`
		`execute`
			Synchronous
			Asynchronous
			Synchronous

Retrofit2	`retrofit2.OkHttpCall`	`request` `enqueue`	Synchronous Asynchronous
Apache HttpClient	`org.apache.http.impl.client.AbstractHttpClient`	`execute`	Synchronous
URLConnection	`libcore.net.http.HttpURLConnectionImpl`	`getContent`	Synchronous

Table 6.2: NetDiet's instrumentation points.

We trace these network events as well as the hook points defined in §5.3.1 expecting to correctly associate executed events with the energy peaks extracted from the power trace.

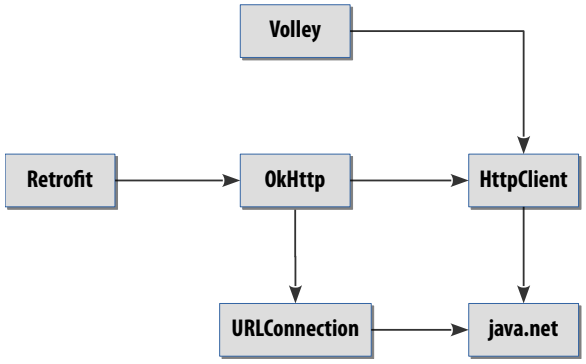


Figure 6.1: Hooked libraries and their dependencies (NetDiet).

Curbing Network-Event Sequences

Meerkat's output is a list of sequence rules decreasingly ordered by their energy support, which supposedly correlates with the potential for saving energy if we interrupt a sequence execution. A developer can use these sequences to identify targets and declare NetDiet policies that are activated once a series of expected network events happen. Following one of Meerkat's principles, the controller in NetDiet blocks network events

according to the confidence of the rules derived from a given sequence pattern, possible using a rate-limiting factor. For synchronous network operations, NetDiet's controller blocks an HTTP request by returning early with an empty response. For asynchronous operations, NetDiet throws an `IOException` or similar exception to simulate network unavailability.

6.2.2 Evaluation

Setup

We conducted experiments on two Samsung Galaxy S4 devices (cf. Table 5.2) running backported versions of the AOSP Lollipop and Marshmallow. We configured NetDiet to observe UI changes and to trace network events only if the app owner is running in the background.

To test the effectiveness of NetDiet, we installed a few mobile apps known for their excessive background network traffic. We base our app selection on industry reports, academic papers [72, 196], and personal experience. To counter bias, we also installed randomly selected popular apps from five categories of the Android Play Store: News, Shopping, Games, Social, and Entertainment. Table 6.3 lists our final installation base. We derived NetDiet policies based on the output from the data-mining engine. We define effectiveness as the extra battery lifetime obtained after applying these runtime policies. For each experiment, we consider the time taken to drain the battery from 80% to 20%, as we have observed the battery decay being more stable within this range. We repeated each experiment three times and report the average lifetime gain and standard deviation. All experiments were conducted in a temperature-controlled lab, with the phones lying on a desk.

Sequence Mining Results

We must first validate that the data-mining engine yields a ranking of sequence rules that highly correlates with the amount of energy they spend. Therefore, we expect that network-related sequences that appear on top have a higher impact on the battery lifetime than those appearing on the bottom.

First we sort the event sequences based on their energy-support. Figure 6.2 (bottom) depicts the distribution of energy-support for all event sequences mined from the execution logs. The final ranking represents a

App Name	Popularity
Google Maps (com.google.android.apps.maps)	1B – 5B
Hangouts (com.google.android.talk)	1B – 5B
YouTube (com.google.android.youtube)	1B – 5B
Google Keep (com.google.android.keep)	10M – 50 M
Google Fit (com.google.android.apps.fitness)	10M – 50M
Google Chrome (com.android.chrome)	1B – 5B
Gmail (com.google.android.gm)	1B – 5B
Google+ (com.google.android.apps.plus)	1B – 5B
CM Updater (com.cyanogenmod.updater)	100K – 500K

(a) System apps

App Name	Popularity
Skype (com.skype.raider)	500M – 1B
ESPN (com.espn.score_center)*	10M – 50M
Weibo (com.sina.weibo)*	10M – 50M
Angry Birds (com.rovio.angrybirds)	100M – 500M
News Republic (com.mobilesrepublic.appy)	10M – 50M
Amazon Shopping (com.amazon.mShop.android.shopping)	50M – 100M
Tapatalk (com.quoord.tapatalkpro.activity)*	10M – 50M
BaconReader (com.onelouder.baconreader)*	1M – 5M
Candy Crush Saga (com.king.candycrushsaga)	500M – 1B

(b) Third-party apps

Table 6.3: Installed apps on device under test (NetDiet). Installation base includes integrated system apps (Google) and popular third-party apps from the Android marketplace. * indicates apps known for high network usage in the background.

long-tailed distribution of sequences, with a rapid decay on support. We pose that a handful of event sequences is responsible for most of the battery drain.

All sequences containing at least one network-related event call are marked with a green circle. The rapid decay on the ranked energy-support of sequences starts after 0.2, thus we set our cutting support threshold to 0.15. As a result, we have four remaining network-relevant sequences. Figure 6.2 (top) lists the events comprising each sequence. We also select three networked sequences from the bottom group. Later, we will compare their battery impact to verify if our distribution hypothesis is correct. Of all ranked sequences, the highest-supported network-related sequence places third on the ranking and traces back to the ESPN app. This sequence is only preceded by two other sequences related to location services.

It is important to note that although we cover networked events from popular third-party libraries, it is possible that applications accessing the Internet via other means could drain the battery without being directly noticed by NetDiet's monitor.

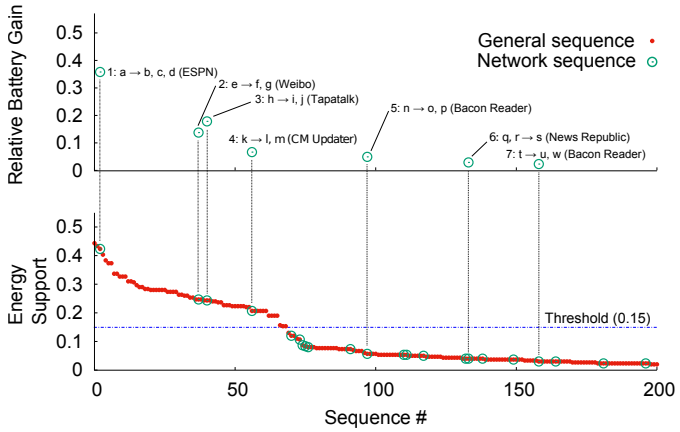


Figure 6.2: Energy support of all mined sequences (bottom) and select network sequences, with corresponding battery life extension (top). We establish a support threshold of 0.15 to focus on the top ranked sequences. For better readability, events names are represented by letter symbols. Table 6.4 serves as a translation guide.

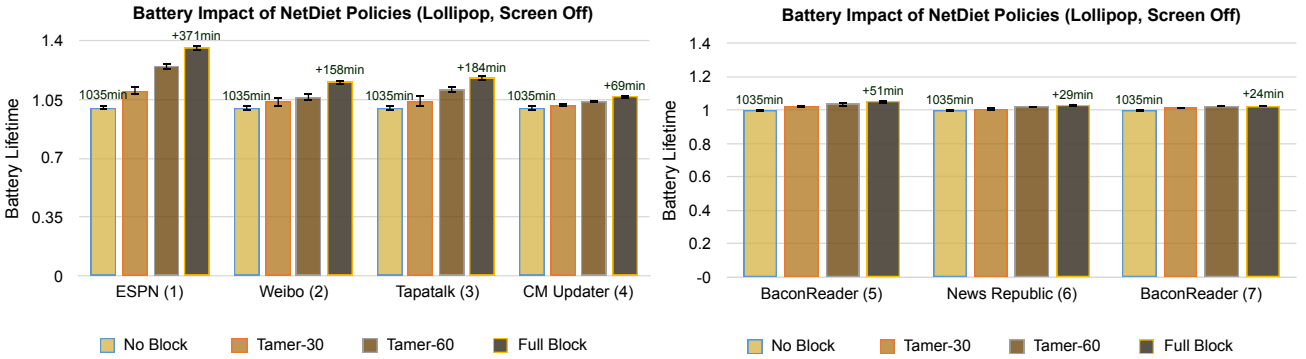
Next, we compare the ranking position of some of the sequence rules with their individual battery-lifetime impact. Because of the time it takes to run a single depletion run, we were not able to assess all network-related sequences. Instead we chose some representative ones for analysis. For each of the four topmost sequences, we derived and installed, in turn, policies that totally block the sequence members and measured the time taken by the battery to decay from 80% to 20% on the device under test. The relative battery lifetime for these four sequences is shown on the top plot of Figure 6.2. We note a very high correlation between the energy-support and the actual battery savings due to blockage of rules, although there is some inversion of positions between the support ranking and battery-gain ranking. This is expected as we are using coarse-grained battery data to infer the energy consumption of concurrent sequences. In the next section, we evaluate flexible policies that guarantee more background utility to affected apps in exchange for lower battery savings.

Lollipop: Screen-Off Events

Using the ranked rules as a guide, we wrote more refined policies that target the topmost four sequences involving network operations. For each sequence rule, we considered two additional types of policies: *Block-30* (event-sequence completion is allowed once every 30 minutes) and *Block-60* (event-sequence completion is allowed once every 60 minutes). *Full Block* corresponds to the total blockage we evaluated in the previous

Symbol	Event Name (Class.Method)	Event Type
a	com.espn.score_center.appwidget.gingerbread.WidgetUpdatingService.onStartJob	Service
b	android.os.HandlerThread.run	Thread
c	com.android.volley.RequestQueue.add	Volley
d	com.android.volley.toolbox.BasicNetwork.performRequest	Volley
e	com.sina.weibo.RemoteRequestService.onStartCommand	Service
f	android.os.HandlerThread.run	Thread
g	libcore.net.http.HttpURLConnectionImpl.getContent	URLConnection
h	com.google.android.gms.internal.zzac.onReceive	BroadcastReceiver
i	android.os.HandlerThread.run	Thread
j	org.apache.http.impl.client.DefaultHttpClient.execute	Apache
k	com.cyanogenmod.updater.service.UpdateCheckService.onStartJob	Service
l	android.os.HandlerThread.run	Thread
m	com.android.volley.RequestQueue.add	Volley
n	com.onelouder.baconreader.NewsExtWidget.UpdateService.onStartJob	Service
o	android.os.HandlerThread.run	Thread
p	okhttp3.RealCall.enqueue	OkHttp3
q	com.cmcm.onews.service.ONewsService	Service
r	com.cmcm.onews.util.push.http.a	AsyncTask
s	org.apache.http.impl.client.DefaultHttpClient.execute	Apache
t	com.onelouder.baconreader.CheckMessageService.onStartJob	Service
u	android.os.HandlerThread.run	Thread
w	okhttp3.RealCall.execute	OkHttp3

Table 6.4: Translation table of symbols to traced events (cf. Figure 6.2).



(a) Top-ranked sequences turned into NetDiet policies.

(b) Bottom-most sequences turned into NetDiet policies.

Figure 6.3: AOSP Lollipop: Relative impact on battery lifetime due to different policy targets and rates. For comparison effect, we show the lifetime improvements for both topmost and bottommost ranked sequences turned into policy targets.

section (cf. Figure 6.2), whereas *No Block* refers to a system running without limiting policies. Figure 6.3a shows the impact of each policy on four apps impacted by the topmost sequences: ESPN, Weibo, Tapatalk, and CM Updater. As we decrease the opportunities for background network I/O, we see a proportional increase of battery lifetime. More importantly, the degree of lifetime improvements highly correlates with the order proposed by our sequence-rule engine (with some variability).

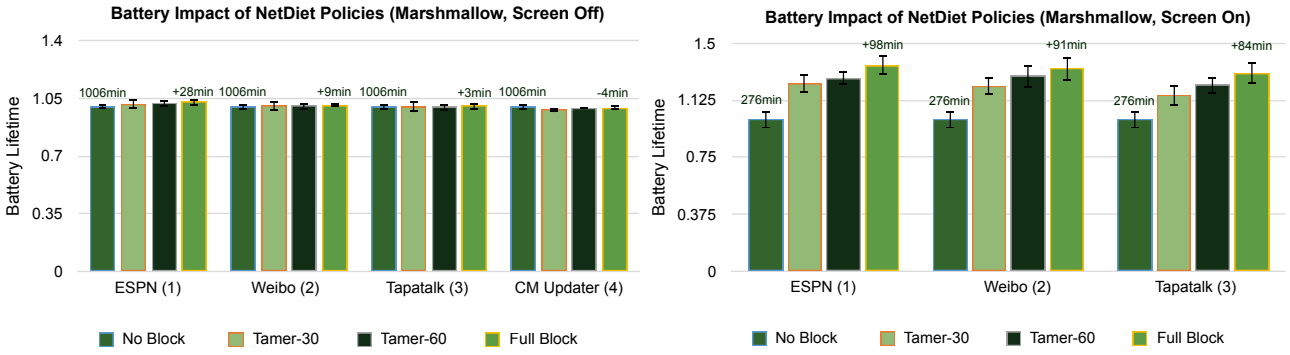
We also consider the battery impact of three networked sequences residing in the tail end of the support-ordered distribution. Figure 6.3b shows that the battery savings from rate-limiting their execution are minimal.

Rule ID	**No Block** (Network KB)	**No Block** (Network Fetches/Pushes)	**Full Block** (Network KB)	**Full Block** (Network Fetches/Pushes)
ESPN (1)	10,352	223	1,424	29
Tapatalk (2)	7,341	131	341	7
Weibo (3)	15,239	60	746	3
CM Updater (4)	3,551	67	0	0
BaconReader (5)	4,835	34	823	14
News Republic (6)	538	7	43	1
BaconReader (7)	4,343	29	4,023	22

Table 6.5: Network data and connection usage for the top-ranked and bottom-most sequences turned into NetDiet policies.

To confirm that the improvement in battery life is a result of judicious application of network-related policies, we report in Table 6.5 the amount of bytes exchanged by the WiFi interface in two cases: (1) with a NetDiet policy that fully blocks target events in the background (Full Block); (2) without any NetDiet active policy (No Block). We also report the number of network fetches and pushes for each case. We used the Android `dumpsys` shell application to obtain network statistics. From the data in Table 6.5, we draw the following conclusions.

- As expected, applying restrictive policies reduced the number of bytes exchanged by all applications.
- On the other hand, a policy does not completely restrict the network interface for a given app. NetDiet, as well as Meerkat and Tamer, intercepts and modifies software behavior at the function level. Therefore, while a target event will be forbidden to send or receive packets, other events belonging to the same application can still access the network without restrictions. With the exception of CM Updater, all the other three apps contemplated with policies had access to the network via other events even when we applied the "Full Block" policy to one of its target events.
- Although tempting, the amount of networked data per application is not a definitive proxy for battery impact. In the case of ESPN and Weibo, the latter consumed close to 30% more network data than the former. Nonetheless, the former opened almost 4× more WiFi connections to synchronize data. ESPN wakes up the WiFi radio more frequently and the tail-state energy of each wake has a major contribution to the higher battery impact.



(a) Battery savings during screen-off discharge.

(b) Battery savings during screen-on discharge.

Figure 6.4: AOSP Marshmallow: Doze Mode inhibits most of the screen-off savings. Still, curbing the same policy target with different rates while the system is idling with the screen on yields positive results.

Marshmallow and Doze Mode

Android Marshmallow introduced two features targeting energy savings in backgrounding: *Doze* and *App Standby* [86]. Doze prevents apps from running background tasks or synchronizing networked data after a device remains idle, stationary, and with the screen off for more than 30 minutes. Maintenance windows permit high-priority tasks (i.e., Google Mobile Services) to run periodically during this state. App StandBy works similarly but only affects apps that are not used very often.

We repeated the Lollipop experiments on Marshmallow. Because *Doze* already curbs standby tasks, NetDiet policies did not have many opportunities to act, thereby they did not yield as much savings as we had seen in Lollipop (Figure 6.4a). We considered an alternative scenario where the screen was kept awake and apps were relayed to the background, thus disabling Doze and App Standby. We want to quantify the energy impact of background tasks when a user is interacting with other apps on her phone. We repeated the experiments, keeping the screen awake using a wakelock (which we purposefully ignored when NetDiet reported it as a high energy consumer). Figure 6.4b shows that NetDiet was effective in extending the battery lifetime (approximately an extra hour of battery life for ESPN, Tapatalk, and Weibo, although we did not see changes for CM Updater). Not surprisingly, the battery gains are much smaller due to the awoken screen and CPU.

6.3 Case Study 2: IODiet

Storage I/O is arguably a major performance hindrance in smartphones [135]. The Android I/O stack contains a set of software and hardware layers used by applications for persistent data management. The stack consists of a DBMS, filesystem, block-device driver, and NAND flash-based storage device. SQLite [31] and EXT4 [222] are the default DBMS and filesystem, respectively.

Recent studies show that the Android I/O stack is not optimized for performance and energy efficiency. Lee and Won were the first to suggest that Android apps generate an excessive amount of synchronous I/O operations, most of which from EXT4 journal writes [144]. Using empirical evidence collected from seven representative Android-app scenarios, Kim et al. [136] demonstrated that up to 75% of all write accesses in Android are SQLite-related and up to 90% of all write counts are synchronous. Jeong et al. found similar results [133]. Improper handling of platform I/O requests can worsen essential usability properties in mobile devices: it can decrease the device's overall performance, shorten NAND flash cell lifetime in eMMC, and decrease energy efficiency. Following the steps of NetDiet, we designed IODiet as a tool to control and test different I/O cadences for mobile apps.

6.3.1 Design and Implementation

IODiet's design and motivation does not differ much from NetDiet's. We focus exclusively on secondary-storage operations, namely read and write. We first investigate how Android apps commonly access storage. Although apps can use Java native I/O classes (e.g., `java.nio` and `java.io`), such practice is not encouraged. Due to security reasons, developers have little freedom to access storage, as apps are sandboxed and only have access to compartmentalized, exclusive data directories[1]. Given that app-state is usually incremental and well structured, app developers are lured into using a SQL database as a means to persist data changes [27]. In cases where a database does not suffice, raw I/O functions are encapsulated with added functionality in third-party libraries. From data gathered from AppBrain and libtastic, three popular storage libraries stood

[1] Access to external storage, such as SD cards, requires special system permission.

Library	Class	Hook Point (Method)	Type
Okio	`okio.Buffer`	`writeTo` `readFrom`	Sync / Async Sync / Async
OrmLite	`com.j256.ormlite.support.AndroidCompiledStatement`	`execSql`	Synchronous

Firebase	`com.google.firebase.provider.FirebaseInitProvider`	`query`
		`insert`
		`update`
		`delete`
			Synchronous
			Synchronous
			Synchronous
			Synchronous

SQLite	`android.database.sqlite.SQLiteStatement`	`execute`
		`executeInsert`
		`executeUpdateDelete`
			Synchronous
			Synchronous
			Synchronous

Table 6.6: IODiet's instrumentation points.

out among developers: Okio (21.53% of popularity), Firebase (31.78%), and OrmLite (14.9%). The latter two libraries encapsulate SQL functionality, whereas the former focuses on file I/O. Similarly to NetDiet, the storage libraries hooked by IODiet build on top of each other (Figure 6.5).

We studied the source code of each library, identified the hook points where storage I/O calls are made, and wrote callbacks to trace execution and possibly block event dispatches (see Table 6.6). We also instrumented the Android SQL classes.

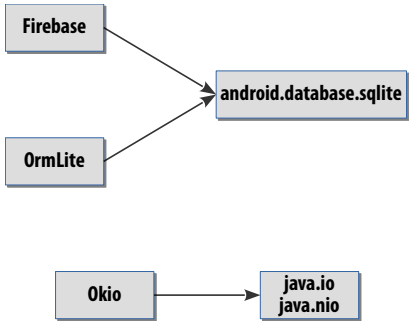


Figure 6.5: Hooked libraries and their dependencies (IODiet).

6.3.2 Evaluation

We used the same device under test (Samsung Galaxy S4) running AOSP Lollipop. To test the effectiveness of IODiet, we installed a few mobile apps known for their excessive background I/O traffic. As a counter to bias we also installed some randomly selected popular apps from the Play Store (Table 6.7). We derived and

installed policies into IODiet based on the output from the data-mining engine. Once again, effectiveness refers to the extra battery lifetime obtained after applying each runtime policy separately.

App Name	Popularity
Google Maps (com.google.android.apps.maps)	1B – 5B
Hangouts (com.google.android.talk)	1B – 5B
YouTube (com.google.android.youtube)	1B – 5B
Google Keep (com.google.android.keep)	10M – 50M
Google Fit (com.google.android.apps.fitness)	10M – 50M
Google Chrome (com.android.chrome)	1B – 5B
Gmail (com.google.android.gm)	1B – 5B
Google+ (com.google.android.apps.plus)	1B – 5B
CM Updater (com.cyanogenmod.updater)	100K – 500K

(a) System apps

App Name	Popularity
Jorte Calendar & Organizer (jp.co.johospace.jorte)	10M – 50M
Subway Surfers (com.kiloo.subwaysurf)	500M – 1B
LINE WebToon (com.naver.linewebtoon)	10M – 50M
Samsung Push Service (com.sec.spp.push)	1B – 5B
BBC News (bbc.mobile.news,ww)	10M – 50M
VivaVideo (com.quvideo.xiaoying)	10M – 50M
Clean Master (Boost&Antivirus) (com.cleanmaster.mguard)*	500M – 1B
LINE B612 - Selfiegenic Camera (com.linecorp.b612.android)	100M – 500M
Bible (com.sirma.mobile.bible.android)	100M – 500M

(b) Third-party apps

Table 6.7: Installed apps on device under test (IODiet). Installation base includes integrated system apps (Google) and popular third-party apps from the Android marketplace. Popularity is measured in terms of number of installs, according to Google Play Store [13]. * indicates an app that is known for high I/O overhead.

Figure 6.6 (bottom) depicts the distribution of energy-support for all sequence rules mined and filtered from the execution logs. Sequences containing at least one storage event call are marked with a red circle. We observe a rapid decay on the ranked energy-support of sequences until 0.05, when the long tail starts. Thus, we set our support threshold to 0.05. As a result, we have three remaining storage-related sequences above the support threshold. Figure 6.6 (top) lists the events comprising each sequence and Table 6.8 serves as a translation guide. We also chose two storage sequences from the bottom group for the sake of comparison.

Just like the case of NetDiet, it is important to note that although we cover storage events from some third-party libraries, it is possible that applications accessing databases or raw files via other means could drain the battery without being directly noticed by IODiet's event logger.

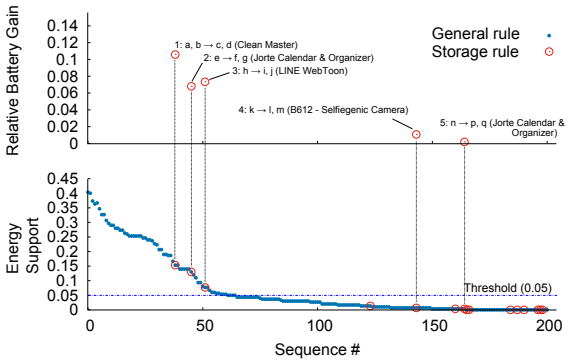
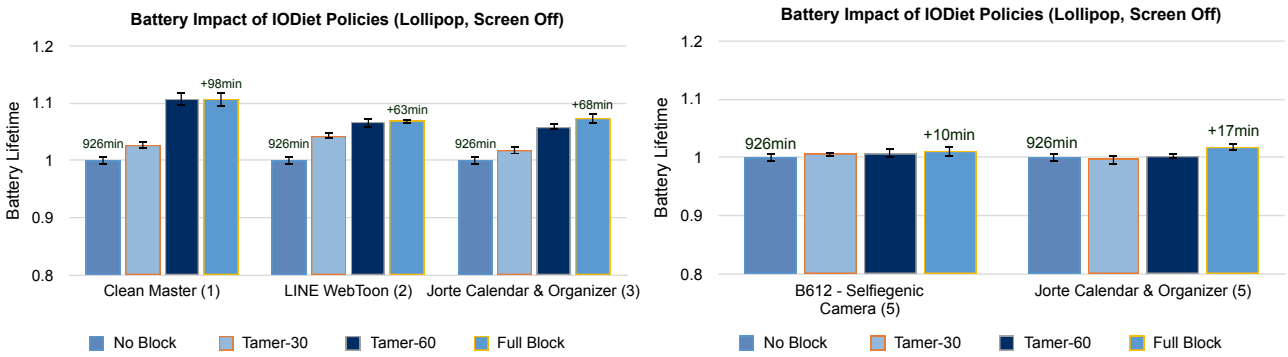


Figure 6.6: Energy support for all mined sequences (bottom) and select storage sequences, with corresponding battery life extension (top). We used a support threshold of 0.05. For better readability, events names are represented by symbols. Table 6.8 serves as a translation guide.

Symbol	Event Name (Class.Method)	Event Type
a	`ks.cm.antivirus.UpdateTimer`	Alarm
b	`ks.cm.antivirus.defend.DefendService`	Service
c	`com.cleanmaster.security.threading.AsyncTask$InternalHandler`	Handler
d	`android.database.sqlite.SQLiteStatement.execute`	SQL
e	`com.jorte.sdk_provider.JorteContentProviderHelperService.onStartCommand`	Service
f	`android.os.Handler.handleMessage`	Handler
g	`android.database.sqlite.SQLiteStatement.executeUpdate`	SQL
h	`com.naver.linewebtoon.download.DownloaderService.onStartCommand`	Service
i	`android.os.HandlerThread.run`	Handler
j	`com.j256.ormlite.support.AndroidCompiledStatement.execSql`	OrmLite
k	`com.liulishuo.filedownloader.services.FileDownloadService.ShareMainProcessService.onStartCommand`	Service
l	`android.os.HandlerThread.run`	Handler
m	`com.j256.ormlite.support.AndroidCompiledStatement.execSql`	OrmLite
n	`com.jorte.sdk_provider.JorteContentProviderHelperService.onStartCommand`	Service
p	`android.os.Handler.handleMessage`	Handler
q	`android.database.sqlite.SQLiteStatement.execute`	SQL

Table 6.8: Translation table of symbols to events (cf. Figure 6.6).

Using the ranked rules as a guide, we wrote more refined policies targeting the three topmost sequences that involve storage operations. The methodology is the same as in the case of NetDiet: for each sequence, we considered four types of policies – *No Block*, *Block-30*, *Block-60*, and *Full Block*. Figure 6.7a shows the impact of each policy on the three apps impacted by the topmost sequences: Clean Master (Boost&Antivirus), Jorte Calendar & Organizer, and LINE WebToon. As we decrease the opportunities for background storage I/O, we see a proportional increase in battery lifetime. Although the degree of lifetime improvements correlates with the sequence order proposed by our sequence-rule engine, the magnitude of the improvements is much smaller. Considering that the topmost storage sequences are ranked far below the topmost general sequences, this is expected. Finally, we consider the impact of applying policies on two storage sequences residing below the support threshold. Figure 6.7b shows that their rate-limiting impact on battery is inconclusive as it is hard



(a) Top-ranked sequences turned into IODiet policies

(b) Bottom-most sequences turned into IODiet policies

Figure 6.7: AOSP Lollipop: Relative impact on battery lifetime due to different policy targets and execution rates. For comparison effect, we show the lifetime improvements for both the topmost and the bottommost ranked sequences turned into policy targets.

to discern whether there are real improvements or whether the positive changes are due to measurement errors.

6.4 Related Work

On Workload Characterization of Networked Applications. To improve the energy efficiency of networked apps, we must first understand how these apps draw power from the battery. Energy consumption can be accounted and apportioned via direct measurement or calibrated power models (see Chapter 2). Using traces from cellular towers, Qian et al. [186] designed a power model based on the residency time of a 3G modem in each possible hardware. By observing the energy impact of different state-machine configurations, the authors optimized the efficiency of radio use. Huang et al. conducted a similar study [124] on 4G LTE networks.

Chen et al. [72] conducted an extensive measurement of energy drain of more than 1,500 smartphones and discovered that: (1) A considerable amount of the total energy drain in a day (45%) occurs during screen-off periods; (2) A minor portion of mobile apps (22.5%) are responsible for a major portion (more than 50%) of the background energy consumption. In another large-scale study, Rosen et al. [196] also verified that although some mobile developers take measures to reduce periodic background traffic, there are many cases of excessive network energy consumption due to wasted screen-off traffic.

Finally, NChecker [134] is a fairly recent tool that detects network-programming defects (NPDs) in mobile

apps arising from intermittent network conditions. NPDs lead to wasted energy, which NChecker can help discover by associating software execution logs with network-connection-condition reports.

On Energy Optimization for Networked Applications. TailEnder [49], Bartendr [205], LoadSense [68], and RadioJockey [44] use different network parameters (e.g., tail-energy overhead, link quality, network load, packet interspacing, etc.) to optimize the scheduling of data transmissions and reduce the energy consumption of common mobile apps. Although they guarantee energy savings, all networked apps are equally affected as the proposed optimizations target the network interface without individually considering the semantics of each application.

Hush [72] is a screen-off optimizer that monitors tasks of various mobile apps, including networking, and automatically identifies and suppresses background activities during screen-off periods that are not useful to the user experience. Usefulness is defined as the correlation between the execution of tasks during a screen-off interval followed by the execution of the same tasks during the next screen-on interval. Procrastinator [192] resembles NetDiet the most. Its control principle is the same – interpose network calls at runtime via code injection. Procrastinator leverages static analysis of bytecode from Windows Phone apps to associate network calls with UI screen elements. Network calls originating from non-visible UI elements are delayed to achieve a 4× reduction in data traffic in exchange for little latency. NetDiet, on the other hand, uses lightweight dynamic tracing to associate power traces with runtime background events, thereby allowing developers to discern and curb the energy-hungriest network events.

On Workload Characterization and Optimization of I/O-Intensive Mobile Applications. The Mobile Storage Analyzer (MOST) [19] is an artifact created by Lee and Won to characterize the performance of storage-block operations on Android. From an in-depth study involving representative smartphone applications [144], Lee and Won suggested an overhaul of the Android storage architecture to accommodate performance optimizations (some of their findings are mentioned in §6.3).

Li et al. proposed a storage energy model for mobile platforms to help developers optimize the energy

requirements of storage-intensive applications [148]. The authors concluded that the two biggest energy consumers related to storage are cryptography and managed-language APIs that provide data privacy and isolation. They also suggest ways to reduce the energy consumption of the storage stack via hardware and software modifications – opt-out of cryptographed storage and specialized hardware to guarantee isolation and privacy.

Jeong et al. [133] described various optimization techniques to improve the SQLite and filesystem performance on Android. Eliminating unnecessary metadata flushes and choosing a different journaling mode in SQL are examples of techniques that can increase the I/O throughput, promote I/O batching, and reduce the energy burden of storage-intensive apps. These modifications are orthogonal to IODiet.

6.5 Conclusion

Mobile app behavior is the main factor behind battery longevity. For instance, background data traffic is not only a threat to a user's limited data plan but also a menace to battery drain. Similarly, mobile software ridden with unoptimized and frequent disk read/write accesses can quickly lead a smartphone to shutdown due to an empty battery. We proposed NetDiet and IODiet, specialized analysis and optimization tools that can help reduce the energy consumption of network- and I/O-heavy Android apps. NetDiet and IODiet leverage Tamer and Meerkat to correlate specific software activity with energy expenditure. This correlation is helpful in deriving controller policies that dictate changes to the runtime behavior of mobile apps. We showed via a few scenarios that both NetDiet and IODiet can deliver a compromise between app functionality and battery longevity that, to the best of our knowledge, other coarse-grained solutions cannot achieve.

Chapter 7

Conclusion

In this dissertation, we explored techniques to analyze, explore, and develop software optimizations for mobile applications in light of limited battery lifetime in mobile devices. Looking from the perspective of how software leverages hardware to execute tasks, we considered how developers can alleviate the impact of task execution on the constrained battery found in smartphones, tablets, and wearable devices. Our contributions are:

- *A software development aid for energy-aware programming.* Application Modes (AM) abstract software functionality into bundles that extend mobile programs and offer users different points on the Pareto frontier between utility and energy savings. With AM, developers can mainly focus on functionality changes that modify the energy profile of an application, while users engage with the system by opting for the bundle that best fits their sense of software usefulness and battery needs. Using AM, developers can expose, at **compile time**, battery optimizations that extend energy proportionality.

- *A binary injection tool for controlling the execution of energy-hungry tasks.* Tamer is an execution controller able to monitor events, rewire program binaries, and change runtime behavior without need of source-code editing. Tamer is a valuable tool to developers longing for a quick way to perform what-if power analysis over different execution paths. Using Tamer, developers can expose, at **runtime**, battery optimizations that extend energy proportionality.

- *A policy-discovery tool for energy controllers.* Meerkat complements TAMER by associating power traces with software events and discovering the most energy-hungry event sequences. Meerkat's output can be used by developers to devise effective policies for TAMER, thus speeding up the discovery of potential energy optimizations.

7.1 Future Work

Based on our initial steps in energy-aware software development and analysis for mobile devices, we envision an integrated suite of tools that supports measurements, analysis, attribution, and visualization of energy consumption by mobile applications. Following the successful example of commercial tools in the realm of performance analysis (e.g., Intel VTune Performance Analyzer [216], CodeXL [112], Snapdragon Profiler [127], etc.,) it is highly desirable to have an easy-to-use, automated and comprehensive system that can live-trace software events, pinpoint and quantify energy bottlenecks of binary code, and readily suggest modifications.

A natural extension to monitoring and profiling energy consumption is to explore automated ways to generate optimized, energy-aware code. This step lends substantial benefits to large-scale software development involving a large code base and numerous input test samples, as is the case of the Android OS. With automatic code generation, we enable software developers and toolchains, such as compilers and runtime, to participate in the energy-aware software development without necessarily imposing expensive runtime energy-saving strategies. One example of a cooperative framework is enDesign [70]. enDesign profiles at function level the energy consumption of running programs and suggests assembly-code optimizations using genetic programming. enDesign uses a guided mutation strategy to create energy-improving program-code mutants based on the observed runtime profile.

enDesign is proven successful against CPU-intensive programs. Its effectiveness lies on a highly accurate energy profiling of and attribution to function calls and critical loop structures. Accuracy is due to linear models modeled after real utilization data sampled at the microsecond scale from hardware performance counters

(e.g., Intel's RAPL interface [197]). In parallel, genetic mutations rely on neutral transformations to the assembly code that preserve the functional equivalence derived from either applying algebraic rules or merging opcodes available in the ISA, or from heuristics that reorders code execution. enDesign evaluates the energy potential of mutated programs using the profiling history from RAPL. Lower-energy mutated programs are kept in the sample population, which serves as source for the next generation of mutated programs.

Adapting genetic programming to general applications that touch multiple hardware subsystems is an interesting albeit challenging idea. First, there is the issue of subpar energy attribution from hardware subsystems to software events. Second, obtaining neutral transformations from high-level code requires a complex expert system that is able to gauge software equivalency of code blocks comprising hundreds or thousands of lines of code (cf. simple assembly operations).

7.2 Open Questions

Along with the direct extensions to our work, other related topics that could be explored in the long term include:

User Involvement in Energy Optimizations. Application Modes bring users into the power-management loop by letting them choose the behavior of the applications running on their smartphones. Although we demonstrated the potential battery savings of adding different modes to an application, we did not assess the user satisfaction with the proposed changes.

Another deficiency in our study is the lack of guidelines on designing new bundles of functionality. Developers are encumbered with the task of bringing forth new modes from the concept stage to implementation and testing. Yet, different users might have different needs that are not necessarily covered by developer-inspired modes. One possible way to involve users in creating modes is to elicit requirements during the design phase of a software project.

As for Tamer and Meerkat, expert users willing to extend the battery life of their devices can, in theory, define policies to programs that do not provide enough power-related knobs. Conventional users, on the other hand,

are mostly at the mercy of software developers. This situation could be reverted if there were open access to effective policies without the need of expertise in power management. A centralized marketplace of curated energy policies for different applications should benefit conventional users interested in extending battery hours. To avoid collusion and biased reviews, a reputation system as well as an incentive mechanism could be applied to the policy repository. XPrivacy [60] is as an example of successful application of this policy-bazaar idea. XPrivacy is a policy-driven controller for the Android OS that prevents applications from leaking privacy-sensitive data. Similarly to TAMER, XPrivacy injects code into binaries to revoke or to allow the execution of operations that can exfiltrate sensitive data from mobile devices. XPrivacy's marketplace offers user-reviewed privacy-related policies targeting popular Android applications.

Impact of Binary Transformations to Software Acceptability. Binary transformations introduced by TAMER generate *relaxed* programs extended with nondeterminism that relaxes their semantics and enables greater flexibility in execution. Going beyond battery savings, relaxation mechanisms, a developer must also ensure that the resulting relaxed program is acceptable. So far we have deemed relaxed programs acceptable if the transformation does not generate crashes nor incur execution changes that are perceivable by the end user. Because this conditioning is subject to what users expect from programs, we resort to the cumbersome task of visually inspecting each generated software individually. It is desirable to streamline this verification process.

Carbin et al. [65] presented language constructs for developing relaxed programs and stating acceptability properties. Acceptability properties include *integrity* properties that the program must satisfy to successfully produce a result and *accuracy* properties that characterize how accurate the generated result must be. Using the Coq Proof Assistant [10], the authors presented proof rules for reasoning about acceptability properties, thereby enabling developers to obtain fully machine-checked verifications of relaxed programs. Adapting Carbin's work to our purpose is not a trivial task, given that, in many cases, we somehow must infer the integrity and accuracy properties from binary programs without access to source code.

-

-

•

Bibliography

[1] https://www.appbrain.com/stats. Accessed on 2016-07-08.

[2] https://www.libtastic.com. Accessed on 2016-07-08.

[3] Advanced Power Management (APM). https://en.wikipedia.org/wiki/Advanced_Power_Management. Accessed on 2016-07-29.

[4] Apache Cordova. http://cordova.apache.org. Accessed: 2016-07-27.

[5] Apple Newton. https://en.wikipedia.org/wiki/Apple_Newton. Accessed: 2016-07-27.

[6] AppMachine. https://www.appmachine.com. Accessed: 2016-07-27.

[7] Bejeweled Blitz. https://play.google.com/store/apps/details?id=com.ea.BejeweledBlitz_na. Accessed on 2016-08-24.

[8] big.LITTLE technology: The future of mobile. https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf. Accessed on 2016-08-12.

[9] Candy Crush Saga. https://play.google.com/store/apps/details?id=com.king.candycrushsaga. Accessed on 2016-09-12.

[10] The Coq proof assistant. https://coq.inria.fr. Accessed on 2016-12-12.

[11] DIY free mobile AppMakr. http://www.appmakr.com. Accessed: 2016-07-27.

[12] The future of mobile subscriptions – Ericsson Mobility Report. https://www.ericsson.com/mobility-report/future-of-mobile-subscriptions. Accessed: 2016-07-27.

[13] Google Play. https://play.google.com. Accessed on 2016-08-23.

[14] Google Services battery drain. https://forums.androidcentral.com/google-nexus-4/302559-google-services-battery-drain.html. Accessed on 2016-08-24.

[15] iPAQ. https://en.wikipedia.org/wiki/IPAQ. Accessed: 2016-07-27.

[16] LG teases Samsung over Galaxy's lack of removable battery. http://www.digitaltrends.com/mobile/lg-tweet-teases-samsung-removable-battery. Accessed on 2016-08-31.

[17] Measuring device power. https://source.android.com/devices/tech/power/device.html. Accessed: 2016-08-12.

[18] MIUI sound recorder. https://github.com/MiCode/SoundRecorder. Accessed on 2016-08-17.

[19] Mobile Storage Analyzer (MOST). http://dmclab.hanyang.ac.kr/sub/main_most.htm. Accessed on 2017-01-12.

[20] Nike+ Run Club. https://play.google.com/store/apps/details?id=com.nike.plusgps. Accessed on 2016-08-24.

[21] NlpWakelock and NlpCollectorWakelock discussion. https://www.reddit.com/r/Android/comments/1rvmlr/nlpwakelock_and_nlpcollectorwakelock_discussion/. Accessed on 2016-08-24.

[22] OsmAnd – offline mobile maps and navigation. https://osmand.net. Accessed on 2016-08-17.

[23] PalmPilot. https://en.wikipedia.org/wiki/PalmPilot. Accessed: 2016-07-27.

[24] Prboom-plus for android. https://github.com/jrgleason/prboom-plus4droid/. Accessed on 2016-08-20.

[25] RoboGuice. https://github.com/roboguice/roboguice. Accessed on 2017-02-23.

[26] Robotium: User scenario testing for Android. https://www.robotium.org. Accessed on 2017-03-31.

[27] Saving data in SQL databases. https://developer.android.com/training/basics/data-storage/databases.html. Accessed on 2017-01-01.

[28] SECuRET LiveStream. https://play.google.com/store/apps/details?id=com.dooblou.SECuRETSpyCam. Accessed on 2017-01-03.

[29] Spycam. com.nikhil.spycam. Accessed on 2016-08-17.

[30] SystemTap. https://sourceware.org/systemtap. Accessed on 2016-08-26.

[31] Using databases in Android: SQLite. https://developer.android.com/guide/topics/data/data-storage.html#db. Accessed on 2016-12-15.

[32] Visual Studio. https://www.visualstudio.com. Accessed on 2016-11-14.

[33] Why RAM boosters and task killers are bad for your Android. https://www.makeuseof.com/tag/ram-boosters-task-killers-bad-android. Accessed on 2016-08-26.

[34] Xposed module repository. https://repo.xposed.info. Accessed on 2016-08-23.

[35] Yuki Abe, Hiroshi Sasaki, Martin Peres, Koji Inoue, Kazuaki Murakami, and Shinpei Kato. Power and performance analysis of GPU-accelerated systems. In *USENIX Workshop on Power-Aware Computing and Systems (HotPower)*, 2012.

[36] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD International Conference on Management of Data*, 1993.

[37] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *IEEE International Conference on Data Engineering (ICDE)*, 1995.

[38] Manish Anand, Edmund B. Nightingale, and Jason Flinn. Ghosts in the machine: Interfaces for better power management. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2004.

[39] Android Developers. Best practices for background jobs. https://developer.android.com/training/best-background.html. Accessed on 2016-08-22.

[40] Android Developers. Managing your app's memory. https://developer.android.com/training/articles/memory.html. Accessed on 2016-08-26.

[41] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[42] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[43] Apple Inc. iOS developer library – background execution. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html. Accessed on 2016-08-26.

[44] Pavan K. Athivarapu, Ranjita Bhagwan, Saikat Guha, Vishnu Navda, Ramachandran Ramjee, Dushyant Arora, Venkat N. Padmanabhan, and George Varghese. RadioJockey: Mining program execution to optimize cellular radio usage. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2012.

[45] AT&T. Application resource optimizer (ARO). https://developer.att.com/application-resource-optimizer. Accessed on 2016-08-22.

[46] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential PAttern Mining using a bitmap representation. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002.

[47] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[48] Rajesh Khrishna Balan, Mahadev Satyanarayanan, So Young Park, and Tadashi Okoshi. Tactics-based remote execution for mobile computing. In *ACM International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2003.

[49] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *ACM Internet Measurement Conference (IMC)*, 2009.

[50] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *ACM Symposium on Operating System Principles and Implementation*, 1999.

[51] Luiz André Barros, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (2nd Edition)*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[52] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12), 2007.

[53] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[54] Yevgen Barsukov. Challenges and solutions in battery fuel gauging, 2004.

[55] Frank Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *ACM SIGOPS European Workshop*, 2000.

[56] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architecture. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4), 2013.

[57] Michael J. Berry and Gordon Linoff. *Data Mining Techniques: For Marketing, Sales, and Customer Support*. John Wiley & Sons, Inc., 1997.

[58] W. Lloyd Bircher and Lizy K. John. Complete system power estimation: A trickle-down approach based on performance events. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2007.

[59] Thomas Bläsing, Leonid Batyuk, Schmidt Aubrey-Derrick, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android application sandbox system for suspicious software detection. In *International Conference on Malicious and Unwanted Software (MALWARE)*, 2009.

[60] Marcel Bokhorst. XPrivacy. https://github.com/M66B/XPrivacy. Accessed on 2016-12-28.

[61] Manjit Borah, Robert Michael Owens, and Mary Jane Irwin. Transistor sizing for low power CMOS circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(6), 1996.

[62] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *ACM SIGMOD International Conference on Management of Data*, 1997.

[63] Niels Brouwers, Marco Zuniga, and Koen Langendoen. NEAT: A novel energy analysis toolkit for free-roaming smartphones. In *ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2014.

[64] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Teharni. Crowdroid: Behavior-based malware detection system for Android. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[65] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[66] Bradley S. Carlson and Roger C. Y. Chen. Performance enhancement of CMOS VLSI circuits by transistor reordering. In *ACM International Design Automation Conference*, 1993.

[67] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIX Annual Technical Conference (ATC)*, 2010.

[68] Abhijnan Chakraborty and Vishnu Navda. Coordinating cellular background transfers using LoadSense. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2013.

[69] Geoffrey Werner Challen, Jason Waterman, and Matt Welsh. IDEA: Integrated distributed energy awareness for wireless sensor networks. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[70] Jie Chen and Guru Venkataramani. enDebug: A hardware-software framework for automated energy debugging. *Journal of Parallel and Distributed Computing*, 96, 2016.

[71] Xiaomeng Chen, Ning Ding, Abhilash Jindal, Y. Charlie Hu, Maruti Gupta, and Rath Vannithamby. Smartphone energy drain in the wild: Analysis and implications. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2015.

[72] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. Smartphone background activities in the wild: Analysis and implications. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2015.

[73] Jerry Cheng, Wong H. Y. Starsky, Hao Yang, and Songwu Lu. SmartSiren: Virus detection and alert for smartphones. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2007.

[74] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, 2007.

[75] David Chu, Aman Kansal, Jie Liu, and Feng Zhao. Mobile apps: It's time to move up to CondOS. In *USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2011.

[76] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: Elastic execution between mobile device and cloud. In *ACM European Conference on Computer Systems (EuroSys)*, 2011.

[77] Ryan Cochran, Can Hankendi, Ayse Kivilcim Coskun, and Sherief Reda. Pack & Cap: Adaptive DVFS and thread packing under power caps. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[78] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *ACM International Conference on Object-Oriented Programming Systems, Languages and Applications*, 2012.

[79] The Nielsen Company. So many apps, so much more time for entertainment. https://www.nielsen.com/us/en/insights/news/2015/so-many-apps-so-much-more-time-for-entertainment.html. Accessed on 2016-08-22.

[80] comScore. The 2015 U.S. mobile app report. https://www.comscore.com/Insights/Presentations-and-Whitepapers/2015/The-2015-US-Mobile-App-Report. Accessed on 2016-10-28.

[81] George Crabtree, Elizabeth Kocs, and Lynn Trahey. The energy-storage frontier: Lithium-ion batteries and beyond. *MRS Bulletin*, 40(12), 2015.

[82] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2012.

[83] Jonathan Crussell, Clint Gibler, and Hao Chen. AnDarwin: Scalable detection of Android application clones based on semantics. *IEEE Transactions on Mobile Computing*, 14(10), 2014.

[84] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making smartphones last longer with code offload. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[85] Gautam Das, King-Ip Lin, Heikki Mannila, Gopal Renganathan, and Padhraic Smyth. Rule discovery from time series. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, 1998.

[86] Android Developers. Optimizing for Doze and App StandBy. https://developer.android.com/training/monitoring-device-state/doze-standby.html. Accessed on 2016-07-04.

[87] Google Developers. Scheduling repeating alarms. https://developer.android.com/training/scheduling/alarms.html. Accessed on 2016-08-21.

[88] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. QUIRE: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011.

[89] Thanh Do, Suhib Rawshdeh, and Weisong Shi. pTop: A process-level power profiling tool. In *USENIX Workshop on Power-Aware Computing and Systems (HotPower)*, 2009.

[90] Mian Dong, Tian Lan, and Lin Zhong. Rethinking energy accounting with cooperative game theory. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2015.

[91] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[92] DU Apps Studio. DU Battery Saver. https://play.google.com/store/apps/details?id=com.dianxinos.dxbs.paid. Accessed on 2016-08-26.

[93] Alfred Dunlop and John Fishburn. TILOS: A posynomial programming approach to transistor sizing. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, 1985.

[94] Prabal Dutta, Mark Feldmeier, Joseph Paradiso, and David Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.

[95] Carla S. Ellis. The case for higher-level power management. In *Workshop on Hot Topics in Operating Systems*, 1999.

[96] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2), 2014.

[97] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. Diversity in smartphone usage. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[98] Adrienne Porter Felt, Steve Chin, Erika an Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[99] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *USENIX Conference on Web Application Development (WebApps)*, 2011.

[100] Oasis Feng. Greenify. https://play.google.com/store/apps/details?id=com.oasisfeng.greenify. Accessed on 2016-08-26.

[101] Niroshinie Fernando, Seng W. Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1), 2013.

[102] Denzil Ferreira, Anind K. Dey, and Vassilis Kostakos. Understanding human-smartphone concerns: A study of battery life. In *International Conference on Pervasive Computing (Pervasive)*, 2011.

[103] Denzil Ferreira, Eija Ferreira, Jorge Goncalves, Vassilis Kostakos, and Anind K. Dey. Revisiting human-battery interaction with an interactive battery interface. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2013.

[104] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *ACM Symposium on Operating Systems (SOSP)*, 1999.

[105] Jason Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *IEEE Workshop on Mobile Computer Systems and Applications*, 1999.

[106] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking energy in networked embedded systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[107] Philippe Fournier-Viger, Antonio Gomariz, Manuel Campos, and Rincy Thomas. Fast vertical mining of sequential patterns using co-ocurrence information. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2014.

[108] Phillipe Fournier-Viger, Roger Nkambou, and Vincent Shin-Mu Tseng. RuleGrowth: Mining sequential rules common to several sequences by pattern-growth. In *ACM Symposium on Applied Computing (SAC)*, 2011.

[109] Martin Fowler. Inversion of Control containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html#ServiceLocatorVsDependencyInjection, 2004. Accessed on 2017-02-23.

[110] Pawan Goal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 1996.

[111] Google Developers. Batterystats and Battery Historian walkthrough. https://developer.android.com/studio/profile/batterystats. Accessed on 2016-08-22.

[112] GPUOpen. CodeXL. https://gpuopen.com/compute-product/codexl. Accessed on 2016-12-12.

[113] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and accurate zero-day Android malware detection. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[114] Dirk Grunwal, Charles B. Morrey III, Philip Levis, Michael Neufeld, and Keith I. Farkas. Policies for dynamic clock scheduling. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[115] Ashish Gupta, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, Thirumalesh Bhat, and Syed Emran. Mining energy traces to aid in software development: An empirical case study. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2014.

[116] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

[117] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *IEEE International Conference on Software Engineering (ICSE)*, 2013.

[118] Shuai Hao, Bin Liu, Suman Nath, J. Halfond, William G, and Ramesh Govindan. PUMA: Programmable UI-automation for large scale dynamic analysis of mobile apps. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

[119] Brett D. Higgins, Jason Flinn, T. J. Giuli, Brian Noble, Christopher Peplin, and David Watson. Informed mobile prefetching. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[120] Henry Hoffman, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[121] Mohammad Ashraful Hoque, Matti Siekkinen, Kashif Nizam Khan, Yu Xiao, and Sasu Tarkoma. Modeling, profiling, and debugging the energy consumption of modern devices. *ACM Computing Surveys*, 48(3), 2015.

[122] Cuixiong Hu and Iulian Neamtiu. Automating GUI testing for Android applications. In *International Workshop on Automation of Software Test (AST)*, 2011.

[123] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[124] Junxian Huang, Feng Qian, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. Screen-Off characterization and optimization in 3G/4G networks. In *ACM Internet Measurement Conference (IMC)*, 2012.

[125] IDC Research Inc. Smartphone OS market share, 2016 Q2. https://www.idc.com/prodserv/smartphone-os-market-share.jsp. Accessed on 2016-12-12.

[126] Mashable Inc. These 8 popular game totally kill your battery life. https://mashable.com/2014/10/01/games-battery-draining. Accessed on 2016-12-12.

[127] Qualcomm Technologies Inc. Snapdragon profiler. https://developer.qualcomm.com/software/snapdragon-profiler. Accessed on 2016-12-12.

[128] Square Inc. Dagger – a fast dependency injector for Android and Java. https://square.github.io/dagger. Accessed on 2017-02-23.

[129] International Technology Roadmap for Semiconductors. ITRS 2.0 publication. https://www.itrs2.net/itrs-reports.html. Accessed: 2016-07-27.

[130] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[131] Dietmar Jannach and Simon Fischer. Recommendation-based modeling support for data mining processes. In *ACM Conference on Recommender Systems (RecSys)*, 2014.

[132] Dietmar Jannach, Michael Jugovac, and Lukas Lerche. Adaptive recommendation-based modeling support for data analysis workflows. In *International Conference on Intelligent User Interfaces (IUI)*, 2015.

[133] Sooman Jeon, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *USENIX Annual Technical Conference (ATC)*, 2013.

[134] Xinxin Jin, Peng Huang, Tianyn Xu, and Yuanyuan Zhou. NChecker: Saving mobile app developers from network disruptions. In *ACM European Conference on Computer Systems (EuroSys)*, 2016.

[135] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage*, 8(4), 2012.

[136] Myungsik Kim, Hu-Ung Lee, and Youjip Won. IO characteristics of modern smartphone platform: Android vs. Tizen. In *International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2015.

[137] Mikkel Baun Kjærgaard, Jakob Langdal, Torben Godsk, and Thomas Toftkjær. EnTracked: Energy-efficient robust position tracking for mobile devices. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2009.

[138] Sven Knispel. BetterBatteryStats. https://play.google.com/store/apps/details?id=com.asksven.betterbatterystats. Accessed on 2016-08-26.

[139] Donald Knuth, James H. Morris, and Vaughan Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2), 1977.

[140] Andreas Lachenmann, Pedro José Marrón, Daniel Minder, and Kurt Rothermel. Meeting lifetime goals with energy levels. In *ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[141] H. Andre Lagar-Cavilla, Niraj Tolla, Eyal de Lara, M. Satyanarayanan, and David O'Hallaron. Interactive resource-intensive applications made easy. In *ACM/IFIP/USENIX International Middleware Conference*, 2007.

[142] Latedroid. Juicedefender. https://play.google.com/store/apps/details?id=com.latedroid.juicedefender. Accessed on 2016-08-26.

[143] Jungseob Lee, Vijay Satisha, Michael Schulte, Katherine Compton, and Nam Sung Kim. Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[144] Kisung Lee and Youjip Won. Smart layers and dumb result: IO characterization of an Android-based smartphone. In *International Conference on Embedded Software (EMSOFT)*, 2012.

[145] Kwong-Sak Leung, Ka-Chun Wong, Tak-Ming Chan, Man-Hon Wong, Kin-Hong Lee, Chi-Kong Lau, and Stephen K. W. Tsu. Discovering protein-DNA binding sequence patterns using association rule mining. *Nucleic Acids Research*, 38(19), 2010.

[146] Ding Li, Shuai Hao, Jiaping Gui, and William G. J. Halfond. An empirical study of the energy consumption of Android applications. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[147] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for Android applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2013.

[148] Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce Worthington, and Qi Zhang. On the energy overhead of mobile storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[149] LifeHacker.com. Android task killers explained: What they do and why you shouldn't use them. http://lifehacker.com/5650894. Accessed on 2016-08-26.

[150] Kaisen Lin, Aman Kansal, Dimitrios Lymberopoulos, and Feng Zhao. Energy-accuracy trade-off for continuous mobile device location. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[151] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: An empirical study. In *Workshop Conference on Mining Software Repositories*, 2014.

[152] Xiaotao Liu, Prashant Shenoy, and Mark D. Corner. Chameleon: Application-level power management. *IEEE Transactions on Mobile Computing*, 7(8), 2008.

[153] Yepand Liu, Chang Xu, S. C. Cheung, and Jian Lü. GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering*, 40, 2014.

[154] David Lo and Siau-Cheng Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2006.

[155] David Lo, Siau-Cheng Khoo, and Limsoon Wong. Non-redundant sequential rules – theory and algorithm. *Information Systems*, 34(4–5), 2009.

[156] Jacob R. Lorch and Alan Jay Smith. Operating system modifications for task-based speed and voltage. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[157] Kai Ma, Xue Li, Wei Chen, Chi Zhand, and Xiaorui Wang. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *IEEE International Conference on Parallel Processing*, 2012.

[158] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In *International Workshop of Software Test (AST)*, 2012.

[159] Anudipa Maiti, Yihong Chen, and Geoffrey Challen. Jouler: A policy framework enabling effective and flexible smartphone energy management. In *EAI International Conference on Mobile Computing, Applications and Services (MobiCASE)*, 2015.

[160] Heikki Mannila, Toivonen, Hannu, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3), 1997.

[161] Marcelo Martins, Justin Cappos, and Rodrigo Fonseca. Selectively taming background Android apps to improve battery lifetime. In *USENIX Annual Technical Conference (ATC)*, 2015.

[162] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. Evaluating the effectiveness of model-based power characterization. In *USENIX Annual Technical Conference (ATC)*, 2011.

[163] Gigaom Media. Google's killer Android L feature: Up to 36% more battery life thanks to Project Volta. https://gigaom.com/2014/07/02/googles-killer-android-l-feature-up-to-36-more-battery-life-thanks-to-project-volta. Accessed on 2016-08-22.

[164] Microsoft. Supporting your app with background tasks (XAML). https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh977056.aspx. Accessed on 2016-08-26.

[165] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffman. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[166] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2012.

[167] Mobile Enerlytics LCC. eStar: Because mobile devices are not mobile if they are plugged in. http://mobileenerlytics.com/estar.php. Accessed on 2016-08-22.

[168] Monsoon Solutions, Inc. Mobile device power monitor manual. https://msoon.github.io/powermonitor/PowerTool/doc/Power%20Monitor%20Manual.pdf. Accessed on 2016-09-01.

[169] Monsoon Solutions Inc. Power monitor. https://www.msoon.com/LabEquipment/PowerMonitor. Accessed: 2016-07-27.

[170] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[171] Suman Nath. ACE: Exploiting correlation for energy-efficient and continuous context sensing. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[172] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *ACM Symposium on Operating System Principles (SOSP)*, 1997.

[173] Adam J. Oliner, Anand P. Iyer, Eemil Lagerspetz, Sasu Tarkoma, and Ion Stoica. Carat: Collaborative energy bug detection for mobile devices. In *USENIX Workshop on Hot Topics in System Dependability (HotDep)*, 2012.

[174] Adam J. Oliner, Anand P. Iyer, Eemil Lagerspetz, Sasu Tarkoma, and Ion Stoica. Carat: Collaborative energy diagnosis for mobile devices. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2013.

[175] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: A system for migrating computing environments. In *ACM Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[176] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *ACM International Joint Conference on Pervasive and Ubiquituous Computing*, 2013.

[177] Sewook Park, Dongwoon Kim, and Hojung Cha. Reducing energy consumption of alarm-induced wake-ups on Android smartphones. In *ACM International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2015.

[178] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging in smartphones: A first look at energy bugs in mobile devices. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2011.

[179] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In *ACM European Conference on Computer Systems (EuroSys)*, 2012.

[180] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *ACM European Conference on Computer Systems (EuroSys)*, 2011.

[181] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel K. Midkiff. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[182] Jian Pei, Jiawei Han, B. Mortazavi-Asl, Jianyong Wang, H. Pinto, Qiming Chen, U. Dayal, and Mei-Chun Hsu. Mining sequential patterns by pattern-growth: the PrefixSpan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(11), 2004.

[183] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 1998.

[184] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2001.

[185] Shyama Charan Prasad and Kumar Roy. Circuit optimization for minimisation of power consumption under delay constraint. In *International Conference on VLSI Design*, 1995.

[186] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. Charactering radio resource allocation for 3G networks. In *ACM Internet Measurement Conference (IMC)*, 2010.

[187] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. TOP: Tail optimization protocol for cellular radio resource allocation. In *IEEE International Conference on Network Protocols (ICNP)*, 2010.

[188] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. Profiling resource usage for mobile applications: A cross-layer approach. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[189] Qualcomm. Trepn Profiler. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler. Accessed on 2016-08-22.

[190] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *IEEE International Symposium on Computer Architecture (MICRO)*, 2009.

[191] Nishkam Ravi, James Scott, Lu Han, and Liviu Iftode. Context-aware battery management for mobile phones. In *IEEE International Conference on Pervasing Computing and Communications (PerCom)*, 2008.

[192] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Chris Rieder. Procrastinator: Pacing mobile apps' usage of the network. In *ACM International Conference on Mobile Systems, Applications, and Service (MobiSys)*, 2014.

[193] Byron Reeves and Clifford Nass. *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*. CSLI Lecture Notes. Center for the Study of Language and Information, Stanford University, 2003.

[194] Martin C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007.

[195] John F. Roddick and Myra Spiliopoulou. A survery of temporal knowledge discovery paradigms and methods. *IEEE Transactions on Knowledge and Data Engineering*, 14(4), 2002.

[196] Sanae Rosen, Ashkan Nikravesh, Yihua Gui, Z. Morley Mao, Feng Qian, and Subhabrata Sen. Revisiting network energy efficiency of mobile apps: Performance in the wild. In *ACM Internet Measurement Conference (IMC)*, 2015.

[197] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2), 2012.

[198] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the Cinder operating system. In *European Conference on Computer Systems (EuroSys)*, 2011.

[199] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[200] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. The case for VM-based Cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4), 2009.

[201] Mahadev Satyanarayanan and Dushyanth Narayanan. Multi-fidelity algorithms for interactive mobile applications. *Wireless Networks*, 7(6), 2001.

[202] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. Edge analytics in the Internet of Things. *IEEE Pervasive Computing*, 14(2), 2015.

[203] SaurikIT, LLC. Cydia Substrate. http://www.cydiasubstrate.com. Accessed on 2016-08-26.

[204] Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Clausen, Oman Kiraz, Kamer A. Yüksel, Seyit A. Camtepe, and Sahin Albayrak. Static analysis of executables for collaborative malware detection on Android. In *IEEE International Conference on Communications (ICC)*, 2009.

[205] Aaron Schulman, Vishnu Navda, Ramachandran Ramjee, Neil Spring, Pralhad Deshpande, Calvin Grunewald, Kamal Jain, and Venkata N. Padmanabhan. Bartendr: A practical approach to energy-aware cellular data scheduling. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2010.

[206] Aaron Schulman, Thomas Schmid, Prabal Dutta, and Neil Spring. Demo: Power monitoring with BattOr. In *ACM International Conference on Mobile Computing and Networking*, 2011.

[207] Aaron Schulman, Tanuj Thapliyal, Sachin Katti, Neil Spring, Dave Levin, and Prabal Dutta. BattOr: Plug-and-debug energy debugging for applications on smartphones and laptops. Technical report, Stanford University, 2016.

[208] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[209] Choonsung Shin, Jin-Hyuk Hong, and Anind K. Dey. Understanding and prediction of mobile application usage for smart phones. In *ACM Conference on Ubiquituous Computing*, 2012.

[210] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[211] Suresh Siddha, Venkatesh Pallipadi, and Arjan Van De Ven. Getting maximum mileage out of tickless. In *Ottawa Linux Symposium*, 2007.

[212] Stelios Sidiroglou, Sasa Misailovic, Henry Hoffman, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011.

[213] Eva Siegenthaler, Yves Bochud, Pascal Wurtz, Laura Schmid, and Per Bergamin. The effects of touch screen technology on the usability of e-reading devices. *Journal of Usability Studies*, 7(3), 2012.

[214] Pieter Simoens, Yu Xiao, Pillai Padmanabhan, Zhuo Chen, Kiryong Ha, and Mahadev Satyanarayanan. Scalable crowd-sourcing of video from mobile devices. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[215] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: A platform for OS-level power management. In *ACM European Conference on Computer Systems (EuroSys)*, 2009.

[216] Intel® Software. Intel® VTune™ Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe. Accessed on 2016-12-12.

[217] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime for perpetual systems. In *ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[218] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology (EDBT)*, 1996.

[219] Pang-Ning Tan, Vipin Kumar, and Jaideep Srivastava. Selecting the right objective measure for association analysis. *Information Systems – Knowledge Discovery and Data Mining*, 29(4), 2004.

[220] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. Who killed my battery?: Analyzing mobile browser energy consumption. In *ACM International Conference on World Wide Web (WWW)*, 2012.

[221] Khai N. Truong, Julie A. Kientz, Timothy Sohn, Alyssa Rosenzweig, Amanda Fonville, and Tim Smith. The design and evaluation of a task-centered battery interface. In *ACM International Conference on Ubiquituous Computing (UbiComp)*, 2010.

[222] Ted Ts'o. Android will be using EXT4 starting with Gingerbread. https://www.linux.com/news/android-will-be-using-ext4-starting-gingerbread. Accessed on 2016-12-15.

[223] UzumApps. WakeLock Detector. https://play.google.com/store/apps/details?id=com.uzumapps.wakelockdetector. Accessed on 2016-08-22.

[224] Narseo Vallina-Rodriguez and Jon Crowcroft. ErdOS: Achieving energy savings in mobile OS. In *ACM Workshop on Mobility in the Evolving Internet Architecture (MobiArch)*, 2011.

[225] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. Towards verifying Android apps for the absence of no-sleep energy bugs. In *USENIX Workshop on Power-Aware Computing and Systems (HotPower)*, 2012.

[226] Po-Han Wang, Chia-Lin Yang, Yen-Min Chen, and Yu-Ming Cheng. Power gating strategies on GPUs. *ACM Transactions on Architecture and Code Optimization*, 8(3), 2011.

[227] Ruowen Wang, William Enck, Douglas Reeves, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M. Azab. EASEAndroid: Automatic policy analysis and refinement for security enhanced Android via large-scale semi-supervised learning. In *USENIX Security Symposium*, 2015.

[228] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reducing CPU energy. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 1994.

[229] Jake Wharton. Butter knife. https://jakewharton.github.io/butterknife. Accessed on 2017-01-15.

[230] Network World. Traditional GPS is dead. long live smartphone GPS. http://www.networkworld.com/article/2994379/mobile-wireless/gps-devices-tom-tom-garmin-vs-smartphone-google-maps-apple.html. Accessed on 2016-12-12.

[231] Qing Wu, Massoud Pedram, and Xunwei Wu. Clock-gating and its application to low power design of sequential circuits. *IEEE Transactions on Circuits and Systems*, 47(103), 2000.

[232] Fengyuan Xu, Yunxin Liu, Qun Li, and Yongguang Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[233] Fengyuan Xu, Yunxin Liu, Thomas Moscibroda, Ranveer Chandra, Long Jin, Yongguang Zhang, and Qun Li. Optimizing background email sync on smartphones. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[234] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *ACM International Conference on Mobile Systems, Applications, and Services*, 2012.

[235] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules form imperfect traces. In *ACM International Conference on Software Engineering (ICSE)*, 2006.

[236] Le Yu, Tao Zhang, Xianpu Luo, and Lei Xue. AutoPPG: Towards automatic generation of privacy policy for Android applications. In *ACM CCS Workshop on Security and Privacy in Smartphones (SPSM)*, 2015.

[237] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *ACM Symposium on Operating System Principles (SOSP)*, 2003.

[238] Wanghong Yuan and Klara Nahrstedt. Practical voltage scaling for mobile multimedia devices. In *ACM Conference on Multimedia (MM)*, 2004.

[239] Mohammed J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1), 2001.

[240] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[241] Lide Zhang, Robert P. Dick, Z. Morley Mao, Zhaoguang Wang, and Ann Arbor. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010.

[242] Lide Zhang, Mark S. Gordon, and Robert P. Dick. ADEL: An automatic detector of energy leaks for smartphone applications. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2012.

[243] Heng Zheng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[244] Heng Zheng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: A unifying abstraction for expressing energy management policies. In *USENIX Annual Technical Conference (ATC)*, 2003.

[245] Xun Zou, Wangsheng Zhang, Shijian Li, and Gang Pan. Prophet: What app you wish to use next. In *ACM International Joint Conference on Pervasive and Ubiquituous Computing*, 2013.