# Power, Performance Modeling and Optimization for Mobile System and Applications

by

Lide Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2013

Doctoral Committee:

Associate Professor Robert Dick, Chair
Associate Professor Jason Nelson Flinn
Associate Professor Z. Morley Mao
Assistant Professor Zhengya Zhang
Professor Peter A. Dinda, Northwestern University

# ACKNOWLEDGEMENTS

This thesis was completed with invaluable help from my advisor, Prof. Robert P. Dick. It is him who led me into research and also guided me through every step during my Ph.D. study. I couldn't have started my research life without him. I'm grateful for his advice and discussions.

I'm also deeply thankful for my collaborators and co-advisors. Prof. Z. Morley Mao has given me great guidance ever since the first work in my thesis. Her knowledge and experience in system area allows her to give me insightful suggestions on both my research directions and the approach I take. Prof. Peter Dinda has also been providing me valuable suggestions in many of my works. His extensive experience with the operating system has helped me to overcome so many technical barriers I ran into during the study. I feel very fortunate to have them helping me.

I would like to express my gratitude to Prof. Jason Flinn and Prof. Zhengya Zhang for serving on my thesis committee. The valuable suggestions they provided make my thesis a more complete and better work. I would also like to thank my colleagues, my friends and my collaborators. David Build and I collaborated on the last piece of work and he contributed tremendous work into the implementation of Panappticon, e.g., the kernel logging framework. The valuable discussion between him and me also inspired me a lot on the other part of the work. Without him, I could never finish the work in such a short time. Mark Gordon and I collaborated on two pieces of my works. I have learned a lot

from Mark on his efficient and solid coding style. PowerTutor would not have such broad impact without him. As my great friends and colleagues, Xuejing He, Yue Liu, Yun Xiang, and Phil Knag have all provided me great suggestions on my work.

I'm also lucky to have my best friends, Xiaoran Tong, Jingsi Xie, and Bingxiao Wu, supporting me all along my Ph.D. study. They have brought me so much joy, regardless of where they were. Also, their attitude to career and life have set great examples for me.

Finally, I would like to thank my family and my boyfriend, Shuyi Chen. They have always been encouraging me and consistently supporting me on every decisions I made. I would not make it without them and this dissertation is dedicated to them.

# TABLE OF CONTENTS

# LIST OF TABLES

**Table**

# LIST OF FIGURES

**Figure**

# ABSTRACT

Smartphone usage has experienced significant growth in the recent years. Despite of its popularity, there is a tension between the increasing demand for smartphone performance, e.g., lower response time, and the limited resource provided by smartphones, in particular energy. Unfortunately, the situation has been made even worse due to two major challenges. On the energy side, software developers do not necessarily understand the energy implication of their design decisions. On the performance side, the traditional approach to optimize performance is not necessarily applicable to mobile device due to the difference in workloads and performance bottlenecks. Combined, these difficulties made balancing between energy and performance for mobile systems and applications even more challenging. As a result, many mobile application, and even those developed by mature companies, can make poor decisions, either on performance or on energy.

My thesis is dedicated to address these challenges by providing a practical, automatic, efficient, and effective framework to help mobile system and application developers to monitor, understand, and optimize the performance and energy of their target designs. My approach consists of three major steps. (1) We first enable developers' understanding of energy implication by providing power models and its construction framework. The provided tool, PowerTutor has demonstrated great value by helping a number of developers to monitor the energy usage of the system and applications. We also enable developer's understanding of performance in the mobile paradigm by providing Panappticon, a lightweight, system-wide, fine-grained event tracing system that automatically identifies

user perceived latency. (2) We then characterize and analyze the real-world smartphone usage scenario by studying traces gathered from PowerTutor and Panappticon. Our study suggests optimization and design guidance for smartphone designers. (3) Motivated by our findings, we proposed technique to optimize the application's energy consumption while maintaining user perceived performance. The diagnosis framework ADEL (Automatic Detector of Energy Leaks) we develop detects and isolates wasted energy resulting from unnecessary network communication. Our study reveals common inefficient design decision in popular applications which were unknown before.

# CHAPTER I

# Introduction

Smartphone usage has experienced significant growth in the recent years. By the year 2012, there are over 100 million smartphone users in the U.S. only [6]. Despite of the increasing popularity of smartphone, there remains a tension between the increasing demand on smartphone performance, e.g., lower response time, and the limited resource provided by smartphones, in particular energy. On one hand, users expect smartphones with good performance and new additional hardware–software features that provide smooth user experience. This results in a dramatic increase in the demand for energy. On the other hand, the available energy provided by smartphone is constrained by battery size and weight and improvements in battery technology have historically been slow.

To ease this tension, we are facing the following challenges.

- *Software engineers lack the necessary understanding of the energy implication of their design decisions.* Designing an energy efficient system framework or application requires that software developers understand the energy implication of their design decisions. Unfortunately, many software developers, even the experienced ones, are trained to develop on general purpose machine and hence have limited experience with energy-constrained portable embedded systems such as smartphones. This lack of knowledge and experience requires us to help them to establish the necessary understanding of energy

first.

- *Designing for performance on mobile devices is different from designing for general purpose PCs and servers.* This is mainly because unlike traditional workloads, mobile workloads are interaction-centric. As a result, traditional performance metrics do not necessarily apply to mobile device. For example, capturing the background task's execution time and reducing it accordingly does not necessarily improve user's experience. Moreover, the difference in workloads results in different performance bottlenecks and hence requires distinctive optimization approaches. Accordingly, we also need to help mobile designers to identify the applicable performance metric and understand the performance bottlenecks.

- *Balancing the trade-off between energy and performance is essential for mobile applications and systems to maintain user satisfaction.* Most if not all mobile applications are interactive applications. User experience with the application and the system can be easily hurt if designers only focus on one aspect in their designs. For example, the battery can drain quickly if the application developers uses an aggressive prefetching scheme to prefetch every link on a web page to shorten the response time. Or users can be annoyed by long respond time if the system developers sets the frequency of the processor speed at the lowest level to save energy even when user's interacting. Therefore, any proposed energy saving technique needs to consider the performance implication and vice versa.

- *Little is known of the representative real-world usage scenario of smartphone.* Despite of the increasing popularity of smartphone, little is known of the representative real-world usage scenario of smartphone. As a result, optimization effort has been spent on components without really understanding the necessity for such optimization. For example, a smartphone equipped with quad-core processor has been newly released without questioning the necessity for such parallelization. This absence of understanding motivates

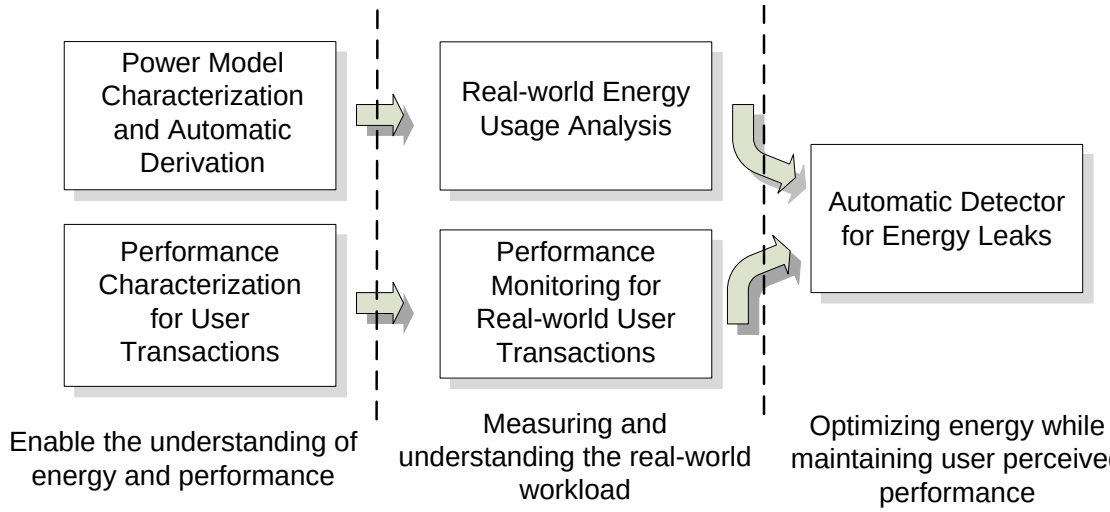



Figure 1.1: Flow chart of the thesis.

us to explore the real-world usage scenario to reveal the real opportunities for optimization.

My thesis is dedicated to address these challenges by providing *a practical, automatic, efficient, and effective framework to help mobile system and application developers to monitor and optimize the performance and energy of their target designs*. Figure 1.1 describes the three major steps I take. I'll elaborate each piece in the following subsections.

## 1.1 Enable the Understanding of Energy Implication and Performance on Mobile Device

We aim at helping developers to design efficient mobile system and applications, both in terms of performance and energy efficiency. To achieve this, the very first step is to enable them to understand both the energy usage and performance of their target designs. For energy, we enable the developers to understand the energy implications of their design decisions by providing a power model and model construction technique. For performance, we help developers to monitor the performance of their designs and detects performance bottlenecks on mobile systems by providing a event-based monitoring framework capturing the user perceived latency.

### 1.1.1 Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphone

We first describe PowerBooter, an automated power model construction technique that uses built-in battery voltage sensors and knowledge of battery discharge behavior to monitor power consumption while explicitly controlling the power management and activity states of individual components. It requires no external measurement equipment. PowerBooter is intended to make it quick and easy for application developers and end users to generate power models for new smartphone variants, which each have different power consumption properties and therefore require different power models.

We also describe PowerTutor, a component power management and activity state introspection based tool that uses the model generated by PowerBooter for online power estimation. PowerTutor is intended to ease the design and selection of power efficient software for embedded systems. Combined, PowerBooter and PowerTutor have the goal of opening power modeling and analysis for more smartphone variants and their users.

### 1.1.2 Panappticon: Event-based Monitoring to Optimize Mobile Application and Platforms

As mentioned before, the interaction-centric nature of mobile systems leads to significant difference between performance monitoring and optimization between mobile device and general purpose PC and servers. This makes the traditional performance metric, e.g., task execution time, and optimization approach inapplicable to mobile device.

We fill this gap with Panappticon, a lightweight, system-wide, fine-grained event tracing system for Android that automatically identifies critical execution paths in user transactions. Panappticon monitors the application, system, and kernel software layers and can identify performance problems stemming from poor application code, underpowered hardware, and negative interactions between otherwise unrelated applications. We car-

ried out a study with 14-user, one-month study. of user-software-hardware smartphone systems. Panappticon allowed us to identify a number of real-world problems that system designers, application developers, and device manufactures can use to improve the user-perceived performance of their products.

## 1.2 Characterization of the Energy Usage and Performance of Real-world Workload

To develop effective optimization strategy for both energy efficiency and performance, it is essential for us to understand the real-world usage scenario to identify bottlenecks. To this end, we deployed both PowerTutor and Panappticon on real users to collect traces, focusing on finding real opportunities to develop optimization strategies.

PowerTutor was released in Android Market. It gathered traces from thousands of real users over a year. One key observation we observed from these traces is that network interfaces (including 3G and WiFi interfaces) is among the top energy hungry hardware components.

Panappticon was deployed on 14 users we recruited from University of Michigan for a month. The deployment result suggests performance bug in applications, system policy inefficiency, e.g., Dynamic Voltage and Frequency Scaling (DVFS), and reveals the impact of architectural decision on user perceived latency, e.g., adding additional core to the processor.

## 1.3 Optimization for Energy Consumption While Maintaining Performance

The trade-off between energy consumption and performance presents a huge challenge for application developers as mentioned before. Any optimization strategy that focuses on one aspect without considering the other can easily hurt user experience.

To overcome this challenge, we define the notion of *Energy leaks*. *Energy leaks* occur when applications use energy to perform useless tasks, a surprisingly common occurrence. They are particularly important for mobile applications running on smartphones due to their energy constraints. Energy leaks are difficult to detect and isolate because their negative consequences are often far removed from their causes. Few tools are available for addressing this problem. We have therefore developed ADEL (Automatic Detector of Energy Leaks). ADEL consists of taint-tracking enhancements to the Android platform. It detects and isolates energy leaks resulting from unnecessary network communication by tracing the direct and indirect use of received data to determine whether it ever affects the user. We profiled 15 applications using ADEL. In 6 of them, energy leaks detected by ADEL and verified by us account for approximately 40% of the energy consumed in communication. We identified four common causes of energy leaks in these applications: misinterpretation of callback API semantics, poorly designed downloading schemes, repetitive downloads, and overly aggressive prefetching.

## 1.4 Thesis Organization

This dissertation is structured as follows.

- Chapter II describes our first two steps from the energy's perspective. It describes the methodology to automatically and manually derive the power model of various of different phone types. It also presents the variation of power models among different phone types. Along with the power model, we also present PowerTutor, the tool that monitors the power consumption of applications and the system. The characterization of real-world PowerTutor traces follows.

- Chapter III presents the first two steps from the performance perspective. It describes our approach to capture and analyze perceived user transaction for mobile systems

based on correlation between fine-grained kernel and user-level events. The traces collected from Panappticon exposes performance inefficiencies in applications and operating system policy. It also reveals the impact of architectural decision on perceived user transaction.

- Chapter IV explains our third step. In Chapter IV, we propose a novel diagnosis framework that isolates and detects energy leaks due to network transmission of various applications. This framework is intended to improve the application's energy efficiency while maintaining user perceived performance.
- Chapter V summarizes related works.
- Chapter VI concludes the thesis.

# CHAPTER II

# Automatic Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones

## 2.1 Introduction

There is tension between the interest in potentially power-hungry smartphone application features and the requirement for low power consumption necessary for long battery lifespans. Designers of smartphone hardware–software platforms have incorporated power-saving features, allowing components to dynamically adjust their power consumptions based on required functionality and performance. However, using these features wisely (or at least avoiding undermining their benefits) requires that software developers understand the implications of their design decisions. Unfortunately, many software developers have limited experience with energy-constrained portable embedded systems such as smartphones. As a consequence, many smartphone applications are unnecessarily power-hungry.

End users have difficulty determining which applications are energy-efficient, and which squander energy; as a result, application users may blame short battery lifespans on operating systems or hardware platforms instead of unfortunate and unintentional software design decisions. Fortunately, application designers have an incentive to develop

energy-efficient smartphone software. Their main barrier is the difficulty of determining the impact of software design decisions on system energy consumption, but that barrier can be overcome.

Researchers have proposed power models for portable embedded systems including Palm [16] and HTC Dream [50]. These power models were derived manually by using a power meter attached to one specific embedded system instance. As a result of the model construction process, the generated power model is at best accurate for one type of embedded system and at worst accurate only for the specific embedded system instance for which it was built. It would require great effort and time to manually generate power models for the wide range of phones now available.

This chapter describes online power model generation techniques. The models produced by these techniques provide accurate, real-time power consumption estimates for power-intensive Android platform smartphone components including CPU, LCD, OLED, GPS, and audio, as well as Wi-Fi and cellular communication components. The proposed system-level power model is based on the influence of the power management and activity states of hardware components on system power consumption. Hardware components are associated with system variables, e.g., LCD brightness, that are subject to introspection and allow estimation of component power consumptions. In additional to system-level power modeling, we describe a technique to model the power consumption of concurrent running applications on multitasking systems.

In this chapter, we show that phones of different types have significant differences in power consumption properties and provide evidence that power consumption differences between individual phones of the same type are negligible for HTC Dream, HTC Magic phones and Nexus One phones. Motivated by this observation and the difficulty of generating power models manually, we propose a battery-based automatic model construction

technique. This technique uses the built-in battery voltage sensors common to modern smartphones. Instead of using a power meter, we use this voltage sensor, and a somewhat complex but automated characterization procedure, to generate a power model.

Finally, we describe a widely used on-line power estimation tool, PowerTutor, that determines system-level and application-level power consumption by using the model generation techniques described in this chapter. PowerTutor has been evaluated on the Android HTC Dream (ADP1), HTC Magic (ADP2) phones and Nexus One phones (N1). Using this tool, we performe the largest-scale in-field data gathering experiment to date to understand smartphone energy consumption. Our tool has been downloaded for more than 66,000 times and it is now in use on more than 800 different smartphones every day. We have been able to make a number of observations, e.g., determining the most energy-hungry hardware components, by studying the resulting anonymized power consumption and activity traces. These observation can be used to understand the opportunities in energy optimization for architects and application developers.

The work described in this chapter makes four main research contributions.

1. We provide manually generated power models for HTC Dream, HTC Magic phones and Nexus One phone. These comprehensive system-level models consider CPU, LCD, OLED, GPS, Wi-Fi, cellular, and audio components (see Section 2.2). This is the first work describing a GPS power model and interestingly, our OLED model differs from the prior model [19]. For components with significant power consumption, we find that power consumption is independent of the states of other components. See Section 2.2 for details.

2. *We measure variation in power consumption properties among phones.* In particular, for the phones we studied, we quantify the (small) variation among multiple instances of the same type of phone, and the (large) variation among different types of phones. See

Section 2.3 for details.

3. We describe a novel automated power model construction technique. This technique uses built-in battery voltage sensors and knowledge of battery discharge behavior to monitor power consumption while explicitly controlling the power management and activity states of individual components. *It requires no external measurement equipment.* See Section 2.4 for details.

4. We study and analyze real-world user traces and suggest optimization opportunities for architects and developers based on our observations. See Section 2.7 for details.

In addition, our work makes a practical contribution: we describe an easy-to-use online power estimation technique that uses the power models described above to determine component-level and application-level power consumption during application execution. A software implementation of this estimator has been released on the Android market. This software tool, which we refer to as PowerTutor, has been used by more than 66,000 people. See Section 2.6 for details.

## 2.2 Power Model

In this section, we will first describe the specific smartphone platform we modeled and the measurement setup. We then explain the system-level power modeling methodology, followed by the detailed description of each hardware component, including CPU, LCD, Wi-Fi, cellular interface, GPS, and audio. We explain the modeling of OLED in more details due to its importance and difference with the previous published OLED model. Finally, we explain the application-level power model.

Table 2.1: Hardware Components for HTC Dream

| Hardware component | Detailed description |
|---|---|
| Processor | MSM7201A chipset, including ARM11 application processor, ARM9 modem, and high-performance DSP |
| LCD Display | TFT-LCD flat glass touch-sensitive HVGA screen |
| Wi-Fi interface | Texas Instruments WL 1251B chipset |
| GPS | A-GPS and standalone GPS |
| Cellular | Qualcomm RTR6285 chipset, supporting GSM, GPRS/EDGE, Dual band UMTS Bands I and IV, and HSDPA/HSUPA |
| Bluetooth | Bluetooth 2.0+EDR via Texas Instruments BRF6300 |
| Audio | Built-in microphone and speaker |
| Camera | 3.2-megapixel camera |
| Battery | Rechargeable lithium-ion battery with capacity: 1,150 mAh |
| Storage | MicroSD card slot |

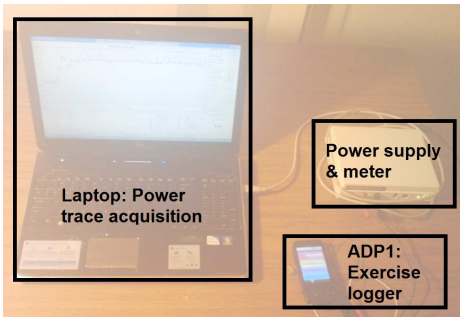



Figure 2.1: Experimental setup for power measurement.

### 2.2.1 Smartphone Hardware Components and Experimental Setup

This section describes power modeling of an Android Dev Phone 1 (ADP1), a version of the HTC Dream mobile phone that permits superuser access. Its hardware components are shown in Table 2.1. We used the Android 1.6 software development kit, which supports both Java and C programming. As Android Dev Phone 2 (ADP2) and Nexus One (N1) phone share most similar hardware components with ADP1, we will describe their system power models in Section 2.3. We describe the OLED display model on Nexus One phone with more details in Section 2.2.4.

We use a Monsoon FTA22D meter [38] for power measurement. The measurement instruments are illustrated in Figure 2.1. The Monsoon meter supplies a stable voltage to

the phone and samples the power consumption at a rate of 5 KHz. During characterization, the ADP1 runs two programs simultaneously. One is a power state exerciser that controls characteristics influencing phone power consumption such as CPU utilization and LCD brightness. This control is not always perfect or precise; we therefore also run a second program to log readings at sufficiently high frequency to capture most changes of system state variables. By using these two programs, we exercise all relevant power states in a relatively short time, and determine the precise system state at any particular time.

### 2.2.2 Modeling Methodology

There are three steps to derive the power model. We will illustrate them individually.

**Selecting hardware components and system variables:** To determine which components need to be considered, we carry out the following experiment for each.

1. Hold the power and activity states of all other components constant.

2. Vary the activity state to extreme values for the components of interest, e.g., set CPU utilization to its lowest and highest values or configure the GPS state to extreme values by controlling activity and visibility of GPS satellites.

For each component, determining the setting that results in extreme power consumption requires some experimentation and knowledge of the component implementation.

Based on these initial experiments, we exclude the components with insignificant impact on the system power consumption, e.g., the SD card. The following components are modeled: CPU and LCD display as well as GPS, Wi-Fi, cellular, and audio interfaces. By measuring the power consumption of the phone when it is at different cross products of extreme power states (e.g., for LCD and CPU, the cross products can be [Full brightness, Low CPU] and [Low brightness, High CPU]), we found that the maximum error resulting from assuming that individual components are independent is 6.27%. This suggests that

a sum of independent component-specific power estimates is sufficient to estimate system power consumption. Note that some hardware components are excluded because their state variables are not visible to the OS, e.g., memory. However, their power consumptions have been accounted for by assigning them to visible (and correlated) components, e.g. memory's power is assigned to CPU.

**Training suites to derive power model:** It is necessary to determine the relationship between each state variable and power consumption for each relevant hardware component. The main idea is to use a set of training programs to change one activity state variable at a time, while keeping all others constant. In each training program, we periodically vary each state variable over its full range. Fixing power states of all other components when exercising one component can reduce measurement noise resulting from state transitions by other components. For example, to determine the relationship between CPU utilization and total power consumption, we fix the CPU frequency and disable the LCD display, as well as the cellular, Wi-Fi, and GPS interfaces. We then use a program to gradually vary the CPU utilization from 0% to 100%. Note that some component power state variables cannot be independently controlled. For example, Wi-Fi and CPU power states are interdependent. To take the influence of interdependent components into account, we also monitor all component power states while exercising the target component. During regression, the power states of all components are considered. In the following subsections, we discuss the implementation of the training programs and the relationship between the power consumption and the corresponding state variables.

**Regression-based approach:** After collecting power traces for hardware components under control of our training software, we use multi-variable regression to minimize the

sum of squared errors for the power coefficient.

$$\begin{pmatrix} P_0 \\ P_1 \\ \cdots \\ P_n \end{pmatrix} = \beta_1 \cdot \begin{pmatrix} U_{01} \\ U_{11} \\ \cdots \\ U_{n1} \end{pmatrix} \ldots + \beta_m \cdot \begin{pmatrix} U_{0m} \\ U_{1m} \\ \cdots \\ U_{nm} \end{pmatrix} + c. \tag{2.1}$$

In this equation, $U_{ij}$ represents system variable $i$ in the $j$th state. $P_j$ is the power consumption when all system variables are in the $j$th state. The inputs for regression are the system variables and the outputs are power consumptions and power coefficients $\beta_i$. Constant $c$ is the minimum system power consumption. Note that Equation 2.1 only represents a linear relationship between system variables and power consumption. However, this is insufficient for some system variables, e.g., processor frequency. In Table 2.2, we use a zero-one indicator associated with a power coefficient to represent the non-linear relationship between frequency and power consumption.

### 2.2.3 Component-level Power Model

**CPU:** CPU power consumption is strongly influenced by utilization and frequency. Varying dynamic, leakage, and peripheral circuit power consumptions invalidate simple cubic frequency–power relationship approximations. Here, we measure the dependence of CPU power consumption on utilization and frequency–voltage settings.

The HTC Dream platform supports two CPU frequencies: 385 MHz and 246 MHz. The corresponding power coefficients are shown in Table 2.2. We consider only the application processor (ARM11); system variables are hidden for the other processor (ARM9), which is dedicated to cellular data and voice services [39]. We model the cellular processor as a part of the cellular interface. The variable $\beta_{CPU}$ shown in Table 2.2 indicates the power difference between active and idle states of the application processor [1].

Table 2.2: HTC Dream Power Model

| Model | $(\beta_{uh} \times freq_h + \beta_{ul} \times freq_l) \times util + \beta_{CPU} \times CPU\_on + \beta_{br} \times brightness + \beta_{Gon} \times GPS\_on + \beta_{Gsl} \times GPS\_sl + \beta_{Wi\text{-}Fi\_l} \times Wi\text{-}Fi_l + \beta_{Wi\text{-}Fi\_h} \times Wi\text{-}Fi_h + \beta_{3G\_idle} \times 3G_{idle} + \beta_{3G\_FACH} \times 3G_{FACH} + \beta_{3G\_DCH} \times 3G_{DCH}$ | | |
|---|---|---|---|
| Category | System variable | Range | Power coefficient |
| CPU | util | 1–100 | $\beta_{uh}$: 4.34<br>$\beta_{ul}$: 3.42 |
| | $freq_l$,$freq_h$ | 0,1 | n.a. |
| | CPU_on | 0,1 | $\beta_{CPU}$: 121.46 |
| Wi-Fi | npackets, $R_{data}$ | 0–∞ | n.a. |
| | $R_{channel}$ | 1–54 | $\beta_{cr}$ |
| | $Wi\text{-}Fi_l$ | 0,1 | $\beta_{Wi\text{-}Fi\_l}$: 20 |
| | $Wi\text{-}Fi_h$ | 0,1 | $\beta_{Wi\text{-}Fi\_h}$: Equation 2.2 |
| Audio | Audio_on | 0,1 | $\beta_{audio}$: 384.62 |
| LCD | brightness | 0–255 | $\beta_{br}$: 2.40 |
| GPS | GPS_on | 0,1 | $\beta_{Gon}$: 429.55 |
| | GPS_sl | 0,1 | $\beta_{Gsl}$: 173.55 |
| Cellular | data_rate | 0–∞ | n.a. |
| | downlink_queue | 0–∞ | n.a. |
| | uplink_queue | 0–∞ | n.a. |
| | $3G_{idle}$ | 0,1 | $\beta_{3G\_idle}$: 10 |
| | $3G_{FACH}$ | 0,1 | $\beta_{3G\_FACH}$: 401 |
| | $3G_{DCH}$ | 0,1 | $\beta_{3G\_DCH}$: 570 |

The CPU training program is composed of a CPU use controller, which controls the duty cycle of a computation-intensive task, and a frequency controller, which writes the system frequency file in the /sys filesystem.

**LCD:** The LCD power model is derived using a training program that turns the LCD on and off and changes its brightness. To simplify modeling, we used 10 uniformly distributed brightness levels.

**GPS:** We consider the influence of the following GPS-related variables on power consumption: mode (e.g., active, sleep, or off), the number of satellites detected, and the signal strength of each satellite. All these variables are logged using the Android Software Development Kit API. To control the GPS state, we use the *requestLocationUpdate* method [1], to make the GPS component switch between sleep and active states. It was

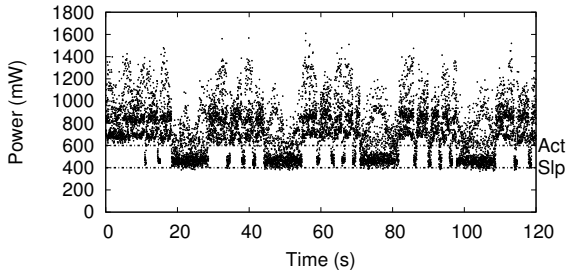



Figure 2.2: Power profile for the current GPS policy.

necessary to change the physical environment of the smartphone to control the number of satellites available and their signal strengths. To this end, we use a conductive hemisphere (i.e., a Faraday Wok) that attenuates radio frequency signals, allowing us to exercise coarse-grained control over the GPS environment. We considered three states: active with many satellites available, active with few satellites available, and sleep. Our measurements indicate that the power consumption depends strongly on whether the GPS component is active or in sleep mode (see Figure 2.2), but has little dependence on the number of satellites available or the signal strength. Considering only the GPS operating mode is sufficiently accurate.

**Wi-Fi:** To derive the Wi-Fi model we consider two network parameters: data rate and channel rate. The Wi-Fi power model is derived by exchanging fixed size (1 KB) TCP packets between the smartphone and a local server. We control the data rate by varying the delay between transmissions from 0 s to 2 s in steps of 0.1 s. These experiments are repeated at uplink channel rates of 11 Mbps, 36 Mbps, 48 Mbps, and 54 Mbps. Repeating the experiment with UDP packets produced similar results.

The Wi-Fi power model depends on four system variables: number of packets transmitted, received per second (*npackets*), uplink channel rate ($R_{channel}$), and uplink data rate ($R_{data}$). Figure 2.3 shows the Wi-Fi power model. The Wi-Fi interface has four power states: *low-power*, *high-power*, *ltransmit*, and *htransmit*. *Ltransmit* and *htransmit* are briefly entered when transmitting data. After sending the data, the card returns to its pre-

vious power state. When transmitting at high data rates, the card is only briefly in the transmit state, i.e., approximately 10–15 ms per second. The time for low-power transmit state is even shorter. The Wi-Fi component power consumption in either transmitting state is approximately 1,000 mW. The *low-power* state is entered when the Wi-Fi interface is neither sending nor receiving data at a high rate. Power consumption in this state is 20 mW. Transition from *low-power* state to *high-power* state happens when more than 15 packets are transmitted or received per second. Interestingly, packet rate, not bit rate, determines the power state. The value of *npackets* must drop to 8 per second to return to the *low-power* state, i.e., the system has hysteresis. In the *high-power* state, the power consumption is 710 mW.

To verify our claim that at a particular channel rate and packet rate, Wi-Fi interface power consumption is independent of bit rate, we repeat the experiments with packet size varying from 0 B to 1 KB, in 100 byte intervals. We observe that the packet size does not influence power consumption given fixed channel and packet rates. However, when the channel rate is low, more time is spent in the very high power consumption transmitting state, given the same amount of data transmitted. The Wi-Fi interface power consumption in *high-power* state is modeled as follows:

$$\beta_{Wi\text{-}Fi\_h} = 710\,\text{mW} + \beta_{cr}\left(R_{channel}\right) \times R_{data} \text{ and} \tag{2.2}$$

$$\beta_{cr}\left(R_{channel}\right) = 48 - 0.768 \times R_{channel}. \tag{2.3}$$

**Cellular:** The cellular interface model is derived by sending UDP packets between a smartphone and a local server via the T-Mobile UMTS 3G network. Packet sizes vary from 10 B to 1 KB. For each packet size, we vary the delay between transmissions from 0 s to 12 s in 0.1 s intervals. Results are similar for TCP packets. The following model does not consider signal strength.

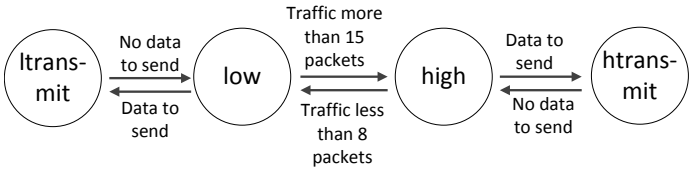



Figure 2.3: Wi-Fi interface power states.

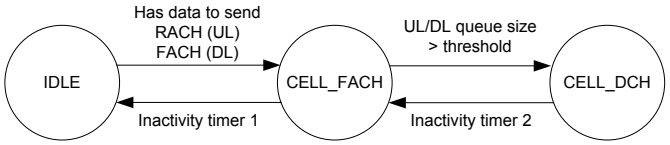



Figure 2.4: 3G interface power states.

The measured data are consistent with the finite state machine based power model shown in Figure 2.4. The model depends on transmit and receive rate (*data_rate*) and two queue sizes. It contains three important states for the communication channel between base station and cellular interface [30].

*CELL_DCH*: In this state, the cellular interface has a dedicated channel for communication with the base station. It can therefore use high-speed downlink/uplink packet access (HSDPA/HSUPA) data rates, resulting in a power consumption of 570 mW for the cellular interface. When there is no activity for a fixed period of time (inactivity timer 2), the cellular interface enters the *CELL_FACH* state.

*CELL_FACH*: In this state the cellular interface shares a communication channel to the base station. It can access the random/forward access (*CELL_RACH*/*CELL_FACH*) common channels. Its data rate is only a few hundred bytes per second. *CELL_RACH* is an uplink channel and *CELL_FACH* is a downlink channel. Cellular interface power consumption in this state is 401 mW. If there is a lot of data to be transmitted, the cellular interface enters the *CELL_DCH* state. Transition from *CELL_FACH* to *CELL_DCH* is triggered by changes in the downlink/uplink queue sizes maintained for these two states in the radio network controller. Our measurements indicate state transition thresholds of 151 bytes for the *uplink_queue* and 119 bytes for the *downlink_queue*. Once either queue size

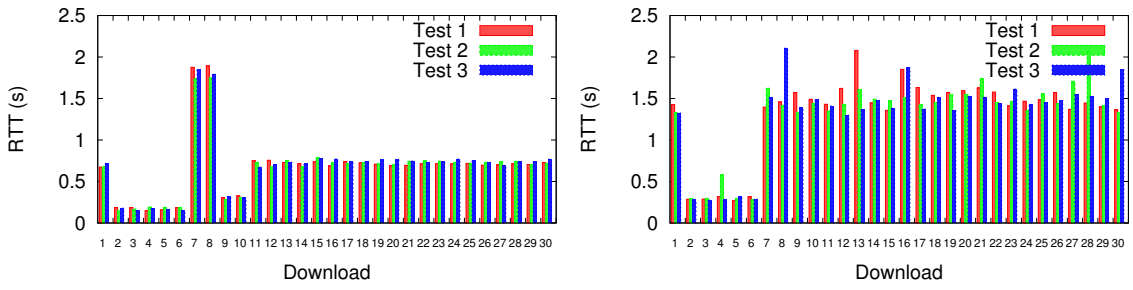



Figure 2.5: (a) TCP handshake RTT. (b) HTTP GET RTT.

exceeds its threshold, *CELL_DCH* is entered. If idle for a sufficient duration (inactivity timer 1), the IDLE state is entered.

*IDLE*: In this state the cellular interface only receives paging messages and does not transmit data. The power consumption is 10 mW.

In order to infer the inactivity durations resulting in state transitions for T-Mobile's UMTS 3G network, we repeatedly download an 80 KB file using HTTP 30 times with a period that increases from 1–29 seconds in one-second intervals, recording the timestamp for each packet. The experiment is repeated 3 times. Before each transmission, the connection is left idle for 30 seconds to allow the cellular interface to enter the idle state. We calculate two Round Trip Times (RTTs) at the beginning of each download. The first RTT is the time between sending out a SYN packet and receiving a SYN-ACK packet during TCP connection set up. The second RTT is the time between sending the HTTP Get request and receiving the first data packet. Figure II.5(a) and Figure II.5(b) show the first and second RTTs calculated for each download. Based on these figures, we can infer the times at which state changes occur due to inactivity.

In Figure II.5(a), the RTTs of download 11, as well as those of subsequent downloads, are equal to the RTT of the first download, which starts from the idle state. Hence the sum of the two inactivity timers is 10 seconds. Figure II.5(a) indicates that the RTTs of downloads 7 and 8 are larger than those of downloads 2–6. This is due to the delay of

the state demotion from *CELL_DCH* to *CELL_FACH*. Figure II.5(b) shows that downloads 2–6 have smaller RTTs than the others; the other downloads experienced delay in state promotion from *CELL_FACH* to *CELL_DCH*. We conclude that inactivity timer 1 is initialized to 6 seconds and inactivity timer 2 is initialized to 4 seconds.

**Audio:** We modeled the audio interface by measuring the power consumption when it is not used, and when an audio file is played at different volumes. The measured data (see Table 2.2) indicate that audio interface use influences power consumption but speaker volume does not. We hypothesize that the increased power consumption during audio output is due to activating a digital signal processor and/or speaker amplifier.

### 2.2.4 OLED Power Model

OLEDs (organic light-emitting diodes) are new display technology that have been used in smartphones produced by HTC and Samsung. Each pixel of OLED display is composed of red, green, and blue sub-pixels, each of which is an independent LED. OLED displays therefore contain no backlights. Consequently, unlike the LCD power model, the power consumption of OLED is not determined by backlight brightness. Instead, it is the sum of the power consumption of every pixel power consumption. As a result, OLED's power consumption depends more strongly on display content than LCD power consumption.

Dong et al. [19] previously proposed an OLED pixel power model. They describe the nonlinear relationship between the power consumption and brightness. Their work also claims that the total display power consumption is the sum of power consumptions yielded by independent sub-pixel (red, green, and blue) power models. Our results appear to contradict theirs, perhaps due to differences on the particular type of display we characterized. We found that modeling sub-pixel power consumptions independently results in 19% error for the N1 handset, as described later. We derived our own OLED model to

refine the error to 4% on average.

For OLED displays, two factors determine a pixel's power consumption: its color and brightness level. A pixel color is specified by an RGB tuple [R, G, B], corresponding to the intensities of red, green, and blue component. Each value in the tuple varies from 0 to a maximum value of 255. Figure II.6(a) shows the relationship of pixel power consumption to these two factors. Figure II.6(a) is obtained by setting all the pixels to a corresponding primary color (red [255, 0, 0], green [0, 255, 0], and blue [0, 0, 255]) while different brightness settings are applied. Figure II.6(a) is obtained by keeping the screen at full brightness while different RGB values are displayed. For example, point *A* represents the power consumption when all pixels on the display are set to dark blue [0, 0, 191]. Note that the power consumption of a black screen has been subtracted from the direct power measurement of the display to eliminate the impact of all other hardware components. From these figures, we observe that the relationship between power consumption and brightness level can be modeled using a linear function, while the relationship between power consumption and RGB value can be best approximated with a quadratic function.

This paragraph describes modeling power consumptions of pixels displaying impure (not just red, green, or blue) colors. Past work [19] claims that OLED pixel power consumption can be modeled as the sum of sub-pixel power consumptions. In contrast, we found that this assumption results in 19% error, perhaps due to characterizing different types of OLED displays. Figure II.6(b) shows the comparison between a model that treats sub-pixels as independent and a model that considers their interdependence. Our measurement results indicate that the intensity of red, green, and blue in white's spectrum is on average 15% lower than the intensity of individual spectrum of red, green, and blue with the same intensity settings. This is consistent with our findings. We speculate that this is a result of manufacturers intentionally dimming impure colors, e.g., white, so that the max-

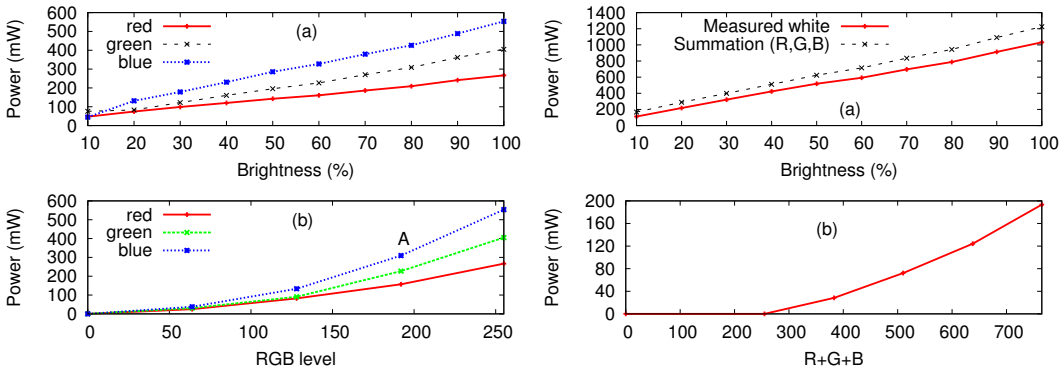



Figure 2.6: (a) OLED power consumption and brightness. (b) Compensation power consumption.

imum intensities of mixed and pure colors are similar. To compensate for this error, we added an additional modulation component to model the difference in power consumption between the sum of sub-pixel power and the measured values. Figure II.6(b) shows the relationship between the modulation component and the sum of sub-pixel intensities.

Based on all the previous observations, we developed the following power model.

$$P_{pixel} = R_{coef} \cdot R^2 + G_{coef} \cdot G^2 + B_{coef} \cdot B^2 \tag{2.4}$$

$$- mod_{coef} \cdot (R + G + B)^2 \text{ and} \tag{2.5}$$

$$P_{screen} = B_{const} + \sum_{i=0}^{N} P_{pixel,i}, \tag{2.6}$$

where $P_{pixel}$ represents the pixel power model. $R, G, andB$ represents the RGB value and $R_{coef}, G_{coef}, andB_{coef}$ are the corresponding coefficients. $mod_{coef}$ represents the coefficient for the modulation component. $P_{screen}$ is the display power model and $B_{const}$ is a constant representing the power consumption when the display is black.

To estimate the power consumption of the display, we need to estimate sub-pixel intensities across the display. The most accurate option would be to consider every pixel. However, the overhead of doing this would be high. We evaluated sampling technique to control overhead while preserving accuracy, and opted for a partially randomized technique to avoid bias when the display contains images with periodic patterns. Specifically,

we divided the display into $n$ equal-sized areas and randomly select one pixel in each area to avoid biasing and aliasing. Section 4.5 shows error and overhead as a function of $n$.

### 2.2.5 Application-Level Power Model

Assigning energy consumption to concurrent running applications is challenging because power state transmissions are sometimes the accumulated result of actions by different applications.

For example, suppose that the Wi-Fi interface transmissions from the low power state to the high power state when transmitting N packets per second. Now imagine that two applications are transmitting at N/2 packets per second, resulting in the Wi-Fi interface entering the high power state. Similarly imagine one application transmits at N packets per second and the other at 9N packets per second. In both these cases, the Wi-Fi interface is in the high power state but it is unclear how to assign power usage to each application. In the first case neither application would trigger the high power state alone so does it make sense to charge them for it? In the second case both applications would trigger the high power state and so should be charged roughly the same amount but what amount should that be?

One possible solution is to divide component power between each application based on the application's workload. This means in the first case each application would be assigned half of the high power state's power and in the second case one would be assigned 1/10 the power and the other 9/10 the power. This solution has the favorable property that the sum of application power usage is equal to the global power usage. This solution is naïve, however, as power usage is not a linear function of transmission rate so it makes little sense to break it up this way. This solution seems even more suspect when we consider that the Wi-Fi interface's power usage is not a one dimensional function.

Instead we need a solution that works independently of each component's power func-

tion and is intuitive to application developers, the major target users of PowerTutor. For each component, we compute the power consumption as if each particular application were running alone. This would mean in case 1, each application would be charged for the lower power Wi-Fi state and in case 2, each application would be charged for the high power state. This loses the nice property that the sum of application power consumptions is equal to the global power consumption (and as these two cases illustrate it is neither an under- nor over-estimate). However, with this definition we can understand the power consumption of an application independently of what else is running. This allows users of PowerTutor to observe similar application-level power characteristics regardless of resource sharing: a useful property for engineers focusing on optimizing particular applications. Note that PowerTutor also reports an accurate system-level power consumption.

There are limitations with this solution; however, we believe it is the best definition available. First, if applications are contending for resources, it is difficult to predict how they would have behaved if they were executing alone. Second, in some cases we see one application invoking another to perform some task. It is unclear how to assign power consumption in this case. On real Android systems, this behavior happens frequently with the media server process. Third, applications that use clever techniques like making transmissions at the same time as other applications don't get rewarded in this scheme. However, there is little that we can do about the first case. Addressing the other two issues requires a high-level understanding of the semantics of the applications involved, which is currently beyond the scope of our tool.

## 2.3 Intra- and Inter-Phone Power Consumption Variation

We previously explained the construction of a power model for an ADP1 phone. In order to determine how general the power model generated for one phone is, we compared models for different instances of the same type of phone, and models for different types of

Table 2.3: Variation of Power Models Among Phones

| Variation (%) | | Intra -ADP1 | Intra -ADP2 | Inter -type |
|---|---|---|---|---|
| CPU | $\beta_{uh}$ | 1.46 | 9.6 | -23.16 |
| | $\beta_{CPU}$ | 9.05 | 9.20 | 33.28 |
| LCD | $\beta_{br}$ | 1.56 | 2.5 | -28.13 |
| Wi-Fi | $\beta_{Wi\text{-}Fi_h}$ | 1.31 | 3.55 | 2.86 |
| | $\beta_{Wi\text{-}Fi_l}$ | 4.89 | 4.86 | -31 |
| Cell | $3G_{DCH}$ | 1.03 | 1.73 | 62.01 |
| | $3G_{FCH}$ | 2.80 | 2.94 | 27.42 |
| GPS | $GPS_{on}$ | 1.35 | 3.01 | -5.12 |
| | $GPS_{sl}$ | 2.48 | 3.82 | -11.50 |
| Audio | $\beta_{audio}$ | 3.31 | 2.57 | -59.37 |

phones. In this section, we characterize two ADP1 phones and four ADP2 (HTC Magic) phones. HTC Magic has the same processor and LCD specifications as the ADP1 (HTC Dream), but a different cellular interface.

Table 2.3 shows the inter- and intra-type power model variation for ADP1 and ADP2 phones. The intra-type variation is the standard deviation normalized by the mean of the sample phones of the same type. The inter-type variation is the difference between the means of the samples for the two types of phones. Note that the power model parameters in the table can also be seen as power measurements for a particular workload, i.e., variation in power model parameters is linearly related to prediction error. For example, for an application using the audio device, we expect to see less than 4% prediction error from using the power model derived for an ADP1 to predict audio device power consumption for another ADP1. These data provide some support for the following conclusions.

First, inter-type variation is significant. Among all the hardware components, the power models for cellular interfaces differ the most, with variation of 62% between ADP1 and ADP2. This result is consistent with data from the Environment Working Group [26], which shows that the ADP2 has greater cellular interface radiation than the ADP1. Interestingly, although ADP1 and ADP2 have the same LCD display specifications, the power

model parameters differ by more than 20%. We speculate that the LCD display in the ADP2 is a more energy-efficient part with the same display quality specifications.

Second, intra-type variation is small. We presently have few intra-type power model samples. To draw a tentative conclusion, we calculated the confidence interval for the intra-type sample variance under the assumption that the distribution of power consumption differences between phones has a Gaussian distribution. With 95% confidence, the maximum intra-type variance is less than 10.4% for all components. This conclusion is tentative because we have not demonstrated that the power consumption difference distribution is Gaussian.

## 2.4 Battery State Based Automated Power Model Generation

We have shown that the variation between power models for different types of phones is significant. This necessitates building a new power model for each type of phone. Current manual measurement based modeling techniques are time-consuming and require access to power measurement instruments. Ideally, it would be possible to quickly and conveniently generate accurate power models for new types of phones without access to special equipment.

We propose a power model generation technique that uses knowledge of battery discharge behavior and the built-in battery voltage sensors in many embedded system, to determine the average power consumption resulting from placing components into different power states. This power characterization technique does not require external power measurement equipment. We now give a brief tutorial on the properties of lithium-ion batteries, which will provide a foundation for explaining the proposed power model generation technique.

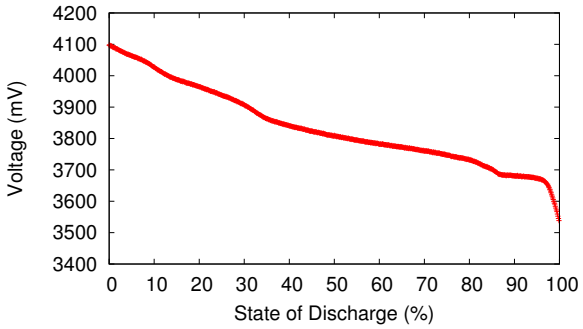



Figure 2.7: Discharge curve of ADP2 lithium-ion battery.

### 2.4.1 Battery Basics

Lithium-ion batteries are popular for portable embedded systems due to their high energy-to-weight ratios, long service lifetimes, and low self-discharge currents.

The voltage of a lithium-ion battery changes during discharge, allowing energy depletion rate (power consumption) to be estimated based on changes to observed voltage. We now explain the reasons for and properties of this voltage change. During discharge, current within the battery is carried by lithium ions (Li+) moving from negative to positive electrodes, through the non-aqueous electrolyte and separator diaphragm [37]. Figure 2.7 shows the discharge curve of the lithium-ion battery in an ADP2 phone with a (relatively low) discharge current of 64.5 mA. The state of discharge (SOD) is the percent of the rated battery energy that has been discharged. As shown in the figure, the discharge curve is monotonically decreasing. Note that both the energy capacity and the discharge curve change with discharge current, temperature, and battery age [12], which may potentially influence the accuracy of the proposed technique, as we will discuss in Section 2.4.2.

The internal impedance of a battery and its load (i.e., a phone) influence its output voltage. A battery can be modeled as a variable resistor in series with a variable voltage source, as shown in Figure 2.4.2. $R_{load}$ is the equivalent resistance of the phone, $R_{int}$ is the internal resistance of the battery, and $V_{int}$ is the internal voltage of the battery. Due to the voltage drop across $R_{int}$, the terminal voltage ($V_{out}$) is lower than $V_{int}$. $V_{int}$ and $R_{int}$ can be

modeled as functions of SOD.

### 2.4.2 Battery State Based Model Generation

The main idea of the battery state based power model generation technique is to use the training software described in Section 2.2 to control phone component power and activity states. The phone components are held in a particular state for a significant period of time and the change in battery SOD is determined using the built-in battery voltage sensor, allowing an estimate of the power consumption for that power management and activity state. At this point, the regression technique described in Section 2.2 can be used to build a power model. One question remains: how can the battery voltage readings be converted into power consumption values? To achieve that, we need to determine the SOD (i.e., total consumed energy) variation within a testing interval based on the sensed voltages:

$$P \times (t_1 - t_2) = E \times (SOD(V_1) - SOD(V_2)), \tag{2.7}$$

where $P$ is the average power consumption in time interval $[t_1, t_2]$, $E$ is the rated battery energy capacity, and $SOD(V_i)$ is the battery state-of-discharge at voltage $V_i$ ($i$ is 1 or 2). The following challenges remain for the proposed technique.

**Determining the SOD based on voltage:** As shown in Section 2.4.1, the present voltage can be used for an inverse lookup of SOD based on the discharge curve. However, there is a potential problem with this idea. The discharge curves of different batteries may vary. Using the same look-up table for all batteries may be inaccurate. We therefore characterize the discharge curve for each battery separately. During characterization, the training software discharges the battery from fully charged to completely discharged states using a constant discharge current, thereby

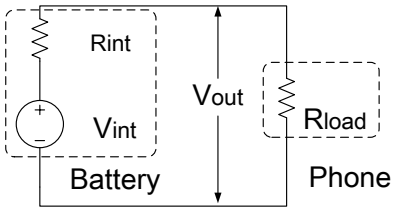



Figure 2.8: Equivalent circuit for battery.

maintaining a linear relationship between SOD and discharging time. A logger runs in background to record the battery output voltage. Note that even for the same phone, the discharge curve may vary with temperature and aging. To eliminate the effect of these external factors, we recommend that characterization be conducted at room temperature (i.e., 73–78°F), in which range the average difference of discharge curve is 4.3%. We acquire the difference by comparing the corresponding voltages at uniformly distributed samples of SOD on the SOD curves under the highest (112.2°F) and lowest temperature (89.6°F) reached by our training suites.

We use a piece-wise linear function to model the non-linear relationship between SOD and battery voltage. To derive the function, we traverse $\{(SOD_1, voltage_1), \cdots, (SOD_n, voltage_n)\}$, which is ordered by increasing voltage values. Each additional SOD and voltage tuple is grouped with the data points on the most recent line segment and linear regression fitting is used. If the maximum error returned by regression is larger than an error threshold, in our case 0.1%, we start a new line segment at the current point.

**Determining the energy capacity $E$:** As shown in Equation 2.7, $E$ is needed to determine power consumption. However, nominal energy capacity may change due to aging and discharge rate. There are two potential solutions to the aging problem. A user with a new battery (e.g., an early adopter wanting to characterize a new type of phone for the first time) can read $E$ from the battery label. For an user with old battery, it would instead be necessary to determine the battery energy based on knowledge of system power consumption in some state, e.g., the maximum CPU power consumption. Note that it would be possible to build a power model for all the components in a new phone without knowing the battery initial energy or the power consumption of any component. However, the power consumptions in this model would be relative to the battery energy, i.e., knowing absolute component power consumptions requires knowledge of the absolute value of

some energy or power consumption number within the model.

Different discharge rates result in different battery energy capacities. We quantified the error resulting from assuming that the battery energy capacity is independent of discharge rate by using the lowest and highest discharge rates the phone can produce during discharge curve characterization. For our ADP2 battery, this results in less than 2.4% error in power consumption estimates.

**The impact of internal resistance:** Due to internal battery resistance, the shape of the discharge curve depends on the discharge current. As a result, even if the internal voltage source has constant voltage, i.e., the battery has the same SOD, the terminal voltage depends on discharge current. Worse yet, the internal resistance depends on SOD. To eliminate the impact of internal resistance, we switch all components of the phone to their low-power modes to minimize discharge current when taking a voltage reading. This minimizes voltage drop across $R_{int}$ in Figure 2.4.2, thereby minimizing the difference between $V_{out}$ and $V_{int}$. We used a voltmeter to verify that $V_{int}$ is within 0.03 V of $V_{out}$ under these conditions.

Some phones that have recently started to appear on the market are equipped with built-in current sensors. This simplifies determining the power consumption for each component power state. However, built-in current sensors are not yet common, so relying on their presence reduces the generality of a power modeling technique.

## 2.5 Power Model Validation

In this section, we evaluate the accuracies of the meter-based (see Section 2.2) and battery-based (see Section 2.4) power models. We first evaluate the meter-based model when running popular applications. Then, we explain the implementation of the proposed battery-based model construction technique and evaluate the resulting model by compar-

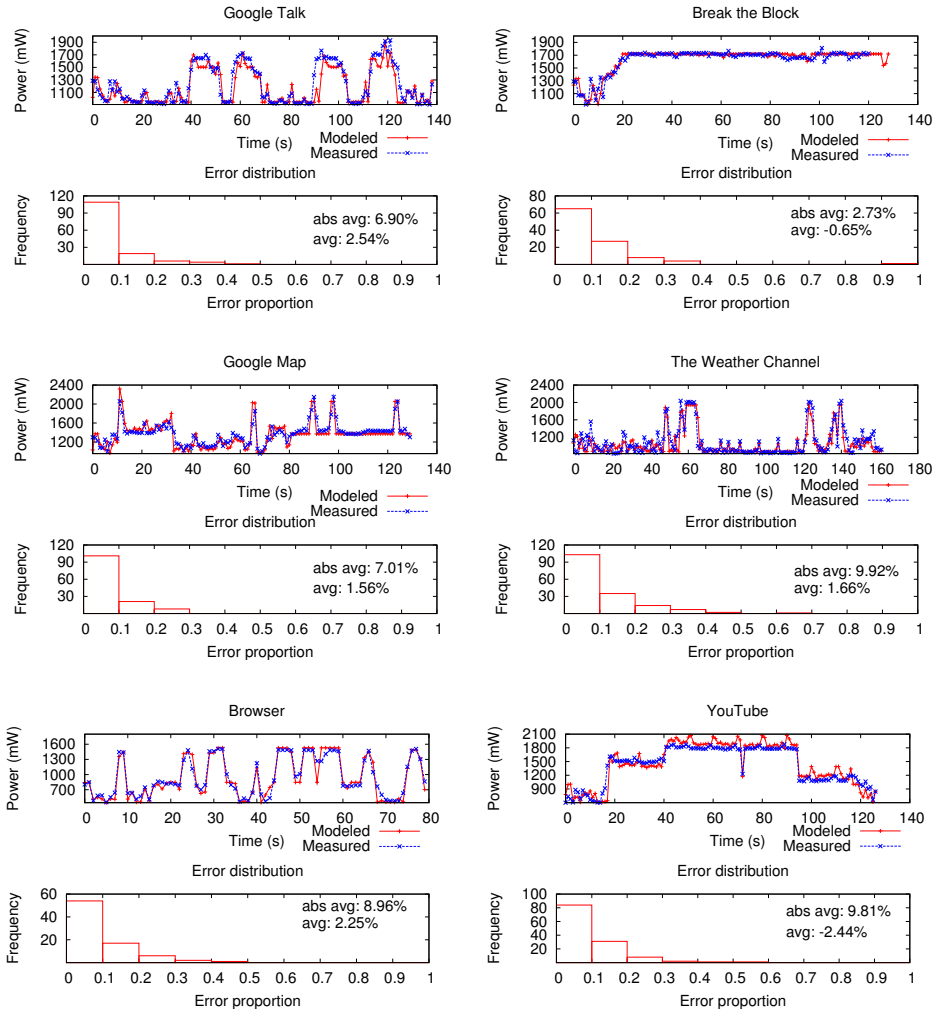



Figure 2.9: Power profiles for selected applications.

ing it with the meter based model.

### 2.5.1 Accuracy Analysis for the Meter-Based System Power Model

We used similar setup as shown in Figure 2.1 for validation. We validated the system-model on HTC Dream and the OLED model on N1 Phone. Note that we validated OLED model individually because of its difference with prior model.

**System-level power model accuracy:** We validated the power model on six popular applications during validation.

- Break the Block: A game that uses CPU, LCD, and Audio.

- Google Talk: An instant message application that uses CPU, LCD, Wi-Fi/3G, and Audio.
- Google Maps: A web mapping application that uses CPU, LCD, Wi-Fi/3G, and GPS.
- The Weather Channel: A weather forecast application that uses CPU, LCD, Wi-Fi/3G, and GPS.
- YouTube: A web-based video sharing application that uses CPU, LCD, and Wi-Fi/3G.
- Browser: The default web browser on Android that uses CPU, LCD, and Wi-Fi/3G.

We repeated the experiment under the following conditions. We used full brightness for Google Maps, Break the Block, and Gtalk; brightness level 102 for YouTube; brightness level 210 for The Weather Channel; and brightness level 36 for Browser. We enabled 3G in Gtalk and used Wi-Fi for all other applications.

To evaluate the accuracy of the model, we used two error metrics. *abs avg* is defined as the average of the absolute values of the errors, i.e.,

$$\left|\frac{\text{measured} - \text{predicted}}{\text{measured}}\right|. \tag{2.8}$$

To estimate the accuracy of the model when estimating the impact of software design on phone battery lifespan, we used another metric, *avg*, i.e.,

$$\frac{\text{measured} - \text{predicted}}{\text{measured}} \tag{2.9}$$

This metric better gauges accuracy predicting the power consumption over long time spans.

Figure 2.9 shows the modeled and measured power consumptions for each application. Error histograms are also shown. The figures show that the average long-term error (*avg*)

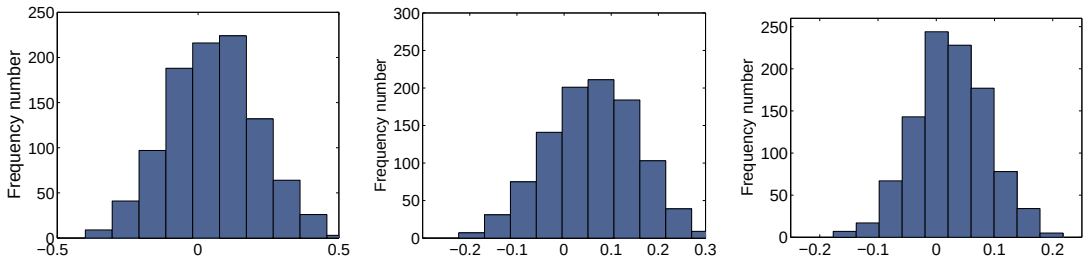



Figure 2.10: Error distribution for LCD. (a) 15 minutes. (b) 30 minutes. (c) 45 mintues.

is less than 2.5% over the application's lifespan and that average error (*abs avg*) is less than 10% for 1 second intervals.

We also measured the power overhead of the on-line power consumption estimation technique. It is only 80 mW, an order of magnitude lower than the power consumption of high-power states of most smartphone components.

**OLED model validation:** During OLED model validation, the N1 runs two programs: a logger to predict OLED power consumption and an exerciser changing the display on the screen. Every benchmark picture is displayed for 2 min. The predicted result is compared with the measured power consumption. Note that the power consumption due to CPU overhead of the logger is excluded from the measurement.

To validate the OLED power model, we did the following two things: (1) validate the pixel power model by sampling the full screen for different screen display, (2) validate the sampling scheme and determine an appropriate sample number by computing the standard deviation in worst case as a function of energy overhead. The worst case happens when the screen is half black and half white because it has the biggest standard deviation of the pixel power consumption. More details will be discussed below.

To validate the pixel power model on frequently displayed pictures with different brightness settings, we did the following two things: (1) we selected 5 default Android wall papers and 3 screen shots of popular applications, e.g., Google Maps, Browser and Pandora, (2) we set the brightness to three different levels including 10%, 50%, and 100%

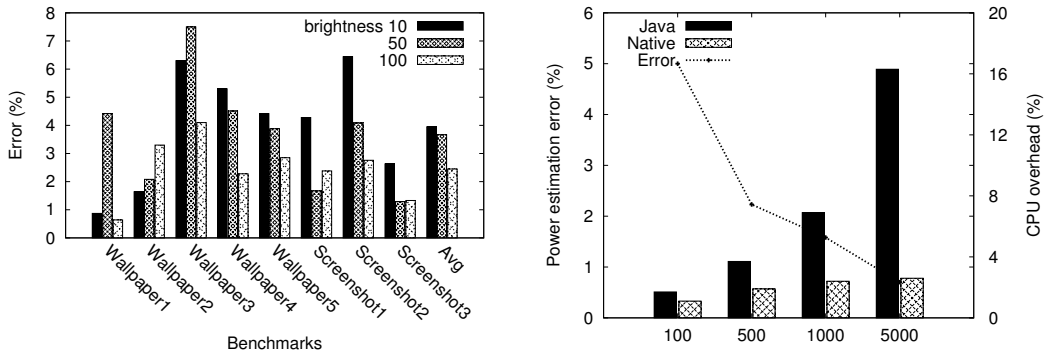



Figure 2.11: OLED model validation and overhead.(a) OLED pixel model validation. (b) Overhead and error for difference sample number.

of full brightness for each picture. Figure II.11(a) shows the percentage error of the predicted power value from measured power value. From Figure II.11(a), we observe that the maximum error is less than 8%, while the average error is less than 4%.

In our implementation we sampled a small set of the pixels to estimate the average power per pixel. The standard deviation of the pixel power mean estimator drawing $n$ samples from a population of $N$ (800×480) from a sample with standard deviation $\sigma$ is given by

$$\sigma\sqrt{\frac{1}{n}(1-\frac{n-1}{N-1})} \tag{2.10}$$

The maximum $\sigma$ is given by $\frac{P_{white}-P_{black}}{4}$ representing a half white half black image and is approximately 298mW.

We had the implementation of sampling both in Java and C for PowerTutor for different handset device. Figure II.11(b) shows how the number of sampled pixels impact the CPU overhead and the error due to sampling. We choose the number of pixels sampled to be 500 (0.13% of the total number of pixels) yielding a standard deviation of our power estimate of 13.4mW in the worst case which we think is reasonable. $\sigma$ for most images being displayed will likely be much lower than 298mW.

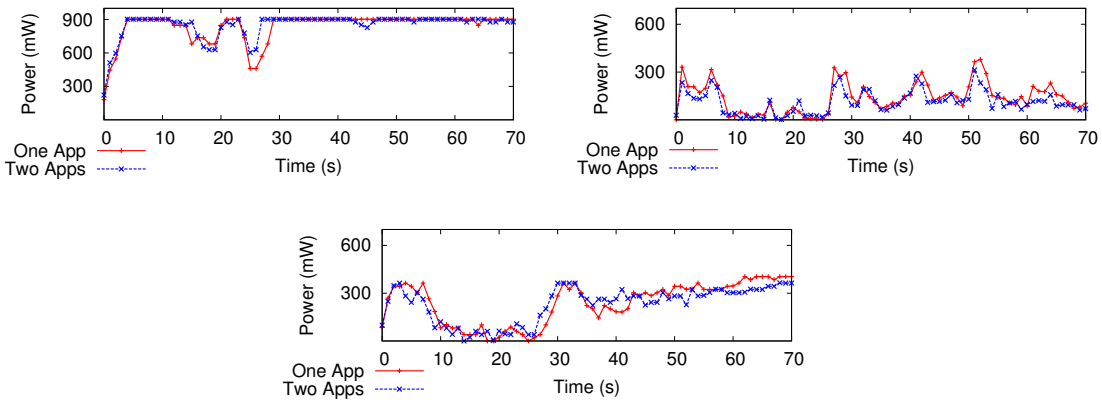



Figure 2.12: Hardware component power for BBC News. (a) 3G power. (b) CPU power. (c) Wi-Fi power.

### 2.5.2 Accuracy Analysis for Application-Level Power Model

To validate application power division we ran an application by itself and with another application running in the background and measured the power usage of each component with PowerTutor in both cases. This required us to repeat the same sequence of user events for the application in different conditions. To accomplish this we recorded events from /dev/input/eventX, exercised the program, and sent the events back to /dev/input/eventX to replay the application. To account for minor changes in the execution across runs it was necessary to repeat the test in each situation 20 times to get a stable mean power each second.

To validate the division of power per application we need to fall back on the definition of application power decided on earlier. That is an application's power consumption as indicated by PowerTutor should be the power usage of that application as if it were running alone. It is important to understand that this definition implies challenging implementation. Multiple applications running at the same time can cause contention for resources in addition to causing internal hardware states to differ, affecting the overall execution of an application. We provide here an analysis of our system as a baseline for future work in the area.

Our results are based on the BBC News application's power consumption when running by itself (One App) and while running with the TuneIn Radio application (Two Apps) on a Nexus One. An execution that visited several news articles was recorded and replayed 20 times for each of the four combinations of 3G/Wi-Fi and one/two apps. The averages of each component's power consumption are recorded in Figure II.12(a), Figure II.12(b), and Figure II.12(c) when BBC News is running by itself and when TuneIn Radio player is streaming a news broadcast at 48 kbps. OLED power consumption has been omitted as the values were nearly identical at all times.

The OLED component had fairly low error and had an average error of 8.25 mW (1.82% of the average power). 3G (Figure II.12(a)) also has low error most of the time with a median error of 0 mW and average error of 28.5 mW (3.22% of the average power). This error could be attributed to differences caused by contention of the radio resource and by the radio being in the dedicated channel (DCH) state for the entire experiment in the two-application case. When our tested application initiated a data transfer in the two-application case it would not go through the normal state transitions as the radio is already in the DCH state.

CPU (Figure II.12(b)) had the second-most average error at 40.2 mW (32.2% of the average power). This error can be attributed to scheduling differences of the operating system since there are other applications running. The exact cause of the differences is difficult to determine. Finally Wi-Fi (Figure II.12(c)) had the highest (absolute) error at 47.6 mW (20.2% of the average power). Interestingly the difference in total Wi-Fi power across the two cases was only 4.0%. A potential source of error could be that the Wi-Fi interface is more likely to go into the high power state when the two applications are running resulting in higher Wi-Fi power consumption initially and lower power consumption after the request has finished.

Overall, this experiment shows that our application-level power model has a good fidelity. The power estimation of the target application when running with another application (Two Apps) tracks the changes in power consumption when it runs alone (One App). This enables the application developers to have a good understanding of what applications are using the most energy and what components they are using.

### 2.5.3 Implementation of the Automatic Battery-Based Model Generation Technique

The power modeling process may be automated as described in Figure 4.2. As described in Section 2.4, constructing the battery discharge curve requires three steps.

1. *Obtain the battery discharge curve for each individual device.* The battery starts in a fully charged state. We characterize the discharge curve individually for each phone.

2. *Determine the power consumption for each component state.* In this step, the state of a single component is varied while other components are kept in low-power states. In order to determine the power consumption for each power state, the battery voltage at the beginning and the end of the power state discharge interval are recorded. The phone is placed in a low-power state immediately before taking a voltage reading to eliminate the impact of the voltage drop across the battery internal resistance. We repeatedly measure voltage for 1 minute, and discharge the battery for 15 minutes between measurements

3. *Perform regression to derive the power model.* After the battery voltage differences for each power state discharge interval are collected, we use them to calculate average power consumption within the 15-minute intervals. We then use regression to generate the power model.

The discharge interval for each power state is difficult to select. Minimizing this duration makes the characterization process more convenient. However, the interval must be long enough for the change in battery voltage to exceed noise. In order to determine the

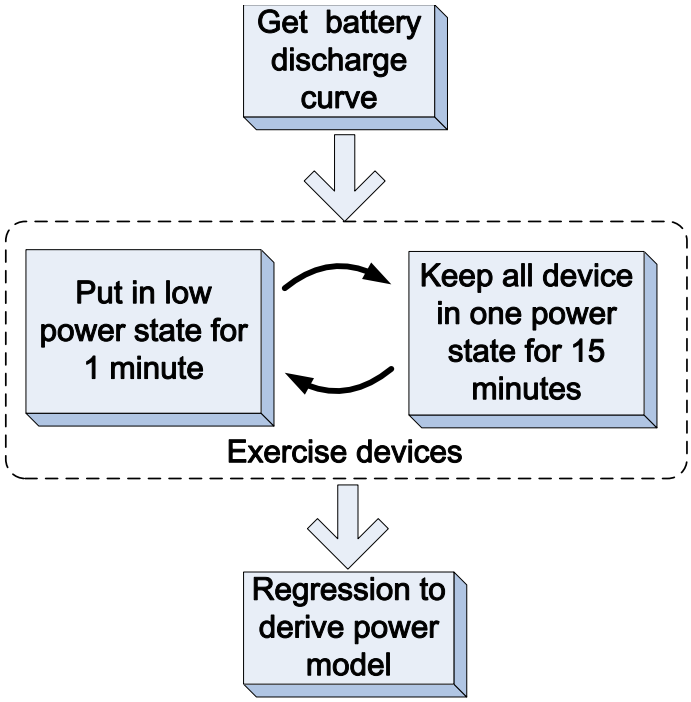



Figure 2.13: Battery SOD based power model construction.

optimal interval, we do statistical analysis of the model error distribution as a function of battery discharge interval per power state. Figure 2.12 shows the error distribution of the LCD model. We did this analysis for all components.

To estimate the error distribution of the battery SOD based technique, one could repeat the entire model construction process many times with different discharge intervals. However, there is a more efficient way to gather the same data. *Bootstrapping* is a technique to treat an initial set of samples as a stand-in for a population and to re-sample from it repeatedly, with replacement. In our experiment, we collect 6 samples at each LCD brightness with a discharge duration of 15 minutes. We then randomly select one sample for each brightness level. By doing regression on these randomly selected samples, we are able to derive an LCD power model. The error is defined as the percentage difference between the newly-derived model and the meter-based model. Repeating this process 1,000 times allows us to determine the error distribution for models generated using 15-minute battery discharge intervals. To determine the distribution for 30-minute intervals, we randomly

select and average two 15-minute sample points without replacement at each brightness level. Note that the experiment was designed to minimize correlation between different samples for the same power state; we repeatedly cycled through all power states six times.

We can draw two conclusions from Figure 2.12. First, the mean of the errors for all discharge intervals deviates from zero by no more than 0.4%. This suggests that, given an adequate battery discharge interval, the battery-based model is as accurate as the meter-based model. Second, the variance of the distribution decreases with longer intervals. For a 45-minute interval, more than 92% of trials have errors less than 10%.

These two conclusions suggest two alternative model-construction strategies. Users can be allowed to choose a trade-off between model construction time and accuracy. We expect that most users would allow a power model to be automatically constructed while they sleep (6.5 hours for a 15-minute battery discharge interval). Another alternative is to have a central web-based system gradually learn the model from samples collected from multiple users. Each user might characterize a phone using a 15-minute battery discharge interval and submit the data. The data for multiple users of the same type of phone could be combined to produce an accurate model. Note that model construction would only need to be done once for a new type of phone, and that automating this process and removing the need for special power measurement equipment would represent a significant improvement on current conditions.

### 2.5.4 Accuracy Analysis for the Battery-Based Power Model

Figure 2.14 shows the error distributions of the battery-based power model using a 15-minute battery discharge interval. The error distribution is generated as described in Section 2.5.3. The box boundaries indicate the 25th and 75th percentiles and the line span indicates the maximum positive and negative errors. The line in the middle of the box is

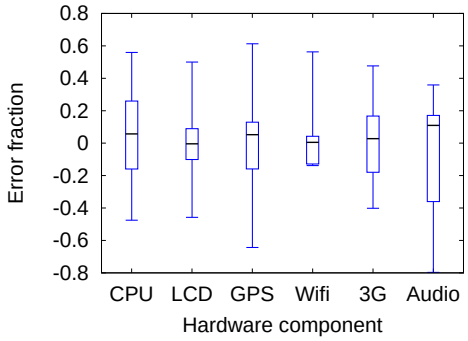



Figure 2.14: Error distributions for components.

the mean of all errors.

## 2.6 Power Estimation Tool

This section describes PowerTutor [44], an online power estimation system that has been implemented for Android platform smartphones. PowerTutor provides accurate, real-time power consumption estimates for power-intensive hardware components including CPU and LCD/OLED display as well as GPS, Wi-Fi, audio, and cellular interfaces. It uses both the system-level and application-level power models generated using the proposed manual (see Section 2.2) or automated (see Section 2.4) characterization techniques. The interface for PowerTutor is shown in Figure 2.15. Figure 2.15(a) shows its application view, on which the list of applications together with their power consumptions are shown. Figure 2.15(b) shows an example display of the power consumption traces of various hardware components. Figure 2.15(c) shows the accumulated energy consumption decomposed by hardware components of the entire phone.

PowerTutor has two main purposes:

- Application developers can use PowerTutor to rapidly, accurately, and conveniently determine the impact of software design changes on power consumption. It provides a time series of power consumption estimates per hardware component of the target application, allowing developers to identify power inefficient behavior, much of which results from

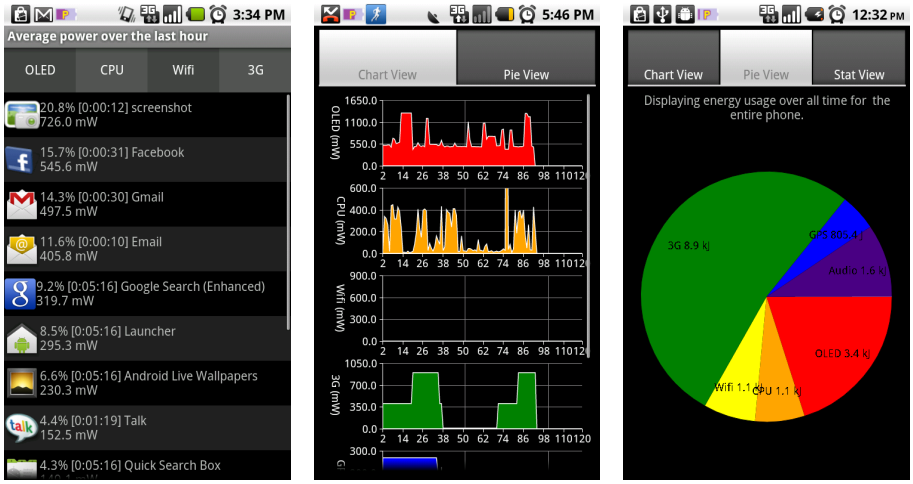



Figure 2.15: PowerTutor interface. (a) Application view. (b) Chart view. (c) Pie view.

unintentional but inappropriate use of smartphone hardware components.

- Smartphone owners can use PowerTutor to determine the power consumption characteristics of competing applications, allowing them to make better informed decisions about which applications to use or buy. Most existing application descriptions and reviews do not mention power consumption. PowerTutor also estimates battery lifespan subject to a particular smartphone owner's actual application usage patterns.

PowerTutor has been released on the Android market and has been used by more than 66,000 people.

### 2.6.1 Comparison to Android's Built-in Meter

Android platform has a built-in battery usage meter. However, the built-in meter was built for a different use case than PowerTutor. Specifically, the built-in meter does not do active monitoring and is intended only to give the user a loose idea of the relative energy consumptions between applications.

We summarized the key differences in Table 2.4. We learned these differences by both using the built-in meter and reading its source. The key benefit of the built-in meter is that it uses nearly no energy itself because it is not actively polling anything. However this

comes at the expense of being able to use accurate models of the phone's various hardware components. Instead the battery meter must make rough estimates of the energy using simple linear models that do not (and could not) take into account intermediate hardware state.

Table 2.4: Comparison Between Android Built-In Power Meter and PowerTutor.

| Feature | Built-in Meter | Power-Tutor |
|---|---|---|
| Active Polling | No | Yes |
| Per Application | Yes | Yes |
| Per Component | Partially | Yes |
| Absolute Power | No | Yes |
| History/Logs | No | Yes |
| Accurate Models | No | Yes |

## 2.7 Understanding optimization opportunities from real-world user traces

As shown in Section 4.5, different applications have varying power consumption, as well as power distribution on hardware components. Therefore, to understand the opportunities for energy optimization, it is necessary for the developers and architects to understand the real-world workload and usage scenarios. In this section, we explore the real-world user traces uploaded by PowerTutor users. We will first describe the data set on which we draw our observations and then present the observations and recommendations for system and application design.

### 2.7.1 Data Set Description

Since PowerTutor was released in November 2009, we have gathered more than 2,000 unique users who have more than one month of traces from more than 50 device types. Given that we currently have the power models for HTC Dream, HTC Sapphire, and Nexus One phones, we focus our power analysis on the users of only these three types. Section 2.5

Table 2.5: Dataset summary

| Type | Num. of users | Time length |
|---|---|---|
| Passion | 58 | 5 - 55 weeks |
| Sapphire | 37 | 5 - 56 weeks |
| Dream | 42 | 5 - 42 weeks |
| Total | 137 | 5 - 56 weeks |

Table 2.6: Data Recorded Each Second by PowerTutor.

| Type | Power | Utils |
|---|---|---|
| CPU | CPU (system/app) power | CPU frequency, CPU usr time, CPU sys time |
| LCD | LCD (system/app) power | screen brightness |
| OLED | OLED (system/app) power | pixel power, screen brightness |
| WiFi | WiFi (system/app) power | WiFi state, packet rate, uplink rate, link speed |
| 3G | 3G (system/app) power | 3G state, operator, byte rate |
| GPS | GPS (system/some app) power | GPS state, number of satellites visible |
| Audio | Audio power | Audio state (on or off) |
| Application | app power | CPU utilization, 3G/WiFi byte rate |

shows the number of users and time length of their traces for each phone type. Section 2.6 summarizes the content we logged in PowerTutor.

### 2.7.2 Key Observations

By analyzing the above data set, we draw the following four observations.

**Display and 3G together dominate the total energy consumption.** Item 2.16 shows the relative energy consumption on hardware components. This figure suggests two conclusions:

1. Display, 3G, CPU, and Wi-Fi device consume more than 99% of the power consumption while the energy consumption of GPS and audio device can be ignored. This is mainly because the time of applications spent using audio and GPS device is significant smaller than the time applications spent on other components. Among all four power hungry hardware components, display and 3G device dominate the total energy consumption: display consumes 45.5% of total energy while 3G consumes 24.8% of total energy. Interestingly, this observation is different from a prior observation [50] which claims that

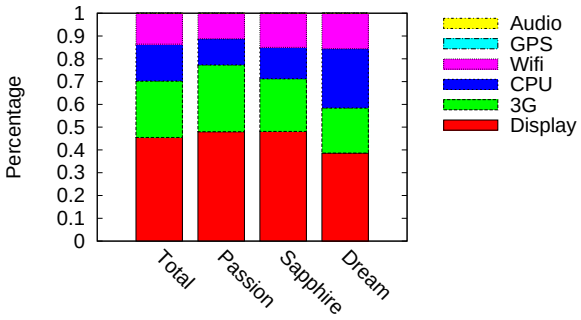



Figure 2.16: Energy decomposition for different phone types.

Table 2.7: Energy Consumed in Idle for 3G and Display.

| Device | Total | Passion (N1) | Sapphire (ADP2) | Dream (ADP1) |
|---|---|---|---|---|
| Display | 40.33% | 44.19% | 33.26% | 42.96% |
| 3G | 38.73% | 40.98% | 38.49% | 33.96% |

display and CPU are the dominant devices. This difference is discussed with greater details in the following observation.

2. The ranking of dominant component depends on phone types. As shown in Item 2.16, display and 3G dominate for both Passion (N1) and Sapphire (ADP2) while display and CPU dominate for Dream (ADP1). Note that the processors' speed of Passion (N1) and Sapphire (ADP2) are faster than Dream (ADP1) phone. We suspect that this difference in CPU speed partially contributes to the difference in ranking: same workload is finished faster on faster processor and therefore results in less energy consumption for CPU.

**Opportunities for energy optimization on display and 3G devices are significant.** After determining that the dominant devices are display and 3G devices, the next question we should ask is how much of their energy consumption can be reduced. To answer this, we start by decomposing the energy further on energy spent when users are actively using the device and when they are not. More specifically, for display, we are interested in the energy consumed when users are not actively interacting with their smartphones. We approximate this by monitoring the CPU utilization: utilization above 5% is considered as

active interaction. For 3G device, we are interested in the energy discharged when the device is staying at high power state without active transmission as explained in Section 2.2.

Item 2.7 summarizes our findings. 40.33% of display energy is consumed when the phone is not actively interacting. This inactive duration can be resulted from two scenarios: (1) users are reading the content in applications between interactions, e.g., reading through the content to find the next button to click. (2) Users are done interacting with the device and the display is about to turn off based on the time-out policy. This result suggests that if either of the above scenario can be shortened, for example, a clearer designed UI or buttons with bigger fonts, so that user can find the next step faster, the battery life can be extended. 38.73% of 3G energy is spent on the tail duration without active transmission. This indicates that most of 3G traffic has short duration so that the tail overhead takes up a significant portion. To eliminate this overhead, one solution for developers is to reduce periodic small traffic transmission and aggregate them into bigger chunk. This suggestion is consistent with prior work [23].

**Current real-world workload rarely keeps the CPU fully utilized.** To understand the optimization opportunity for CPU, we obtain the CPU utilization histogram of all samples for all traces. The four figures on the left in 2.17 summarize the result. This result leads to two observations: (1) the processor spends most of its time idling or doing non-intensive workloads, i.e., the processor is rarely fully utilized. This suggests a very counter-intuitive conclusion: processor nowadays is fast enough to handle current workload. That is to say, we do not need a faster processor for performance purpose. Note that this statement is not true if any the following three assumptions is true: future workloads evolves and therefore has more CPU intensive workloads; instantaneous CPU utilization distribution is not the same as 1-second averaged CPU utilization distribution; although the processor is rarely highly utilized, it is critical for users' experience when it is highly

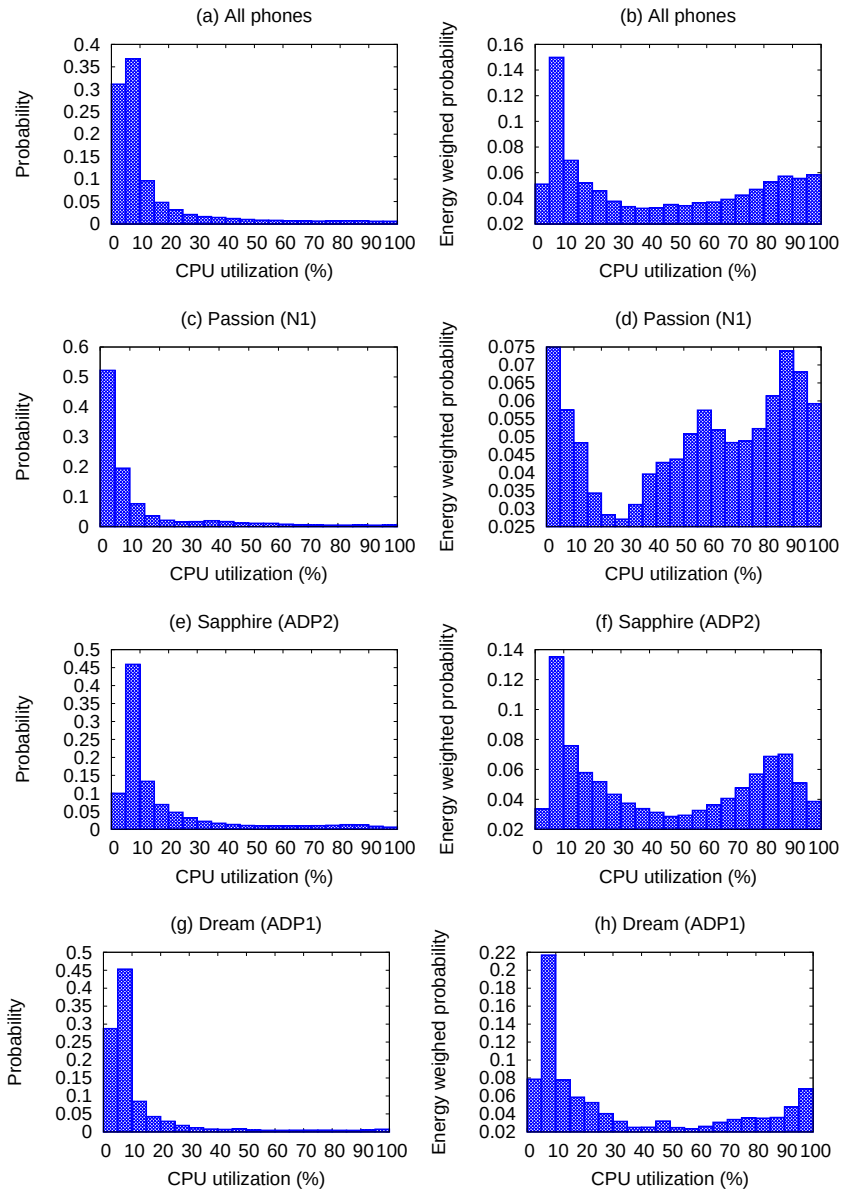



Figure 2.17: (left) Histogram of CPU utilizations. (right) Energy weighted histogram of CPU utilizations.

utilized. (2) With processor speed improves, the histogram shifts toward left. For example, with Dream (ADP1) and Sapphire (ADP2), the spikes appear between 5%-10% utilization, while with Passion (N1), the spike appears between 0%-5%. This shift suggests the improvement of processor speed, workload that used to run with a higher utilization takes smaller utilization with a faster processor.

**Processor architects need to pay attention to optimize the energy efficiency when the processor is almost fully utilized and almost idle.** The right four figures in 2.17

demonstrate the energy weighted histogram of CPU utilization. Each bucket represents the energy spent in that utilization bucket instead of the total count of samples in the bucket. For example, the first bucket in 2.17(d) represents that 7.5% of energy is spent when the CPU is 0%-5% utilized. These four figures also lead to two observations: (1) despite that processors are rarely highly utilized, the energy consumed when they are highly utilized cannot be ignored, especially for the fastest phone, Passion. That is to say, processor architects should focus on optimizing energy efficiency on both the high and low utilization ends for the processor instead of paying their attention on every utilization level. (2) Energy efficiency of high utilization becomes more important with a faster processor. This is because the percentage of energy consumed when the CPU is highly utilized increases with the maximum frequency increases, e.g., 36% of CPU energy is consumed for Passion when the utilization is above 70% while only 33% and 25% of CPU energy is consumed for Sapphire and Dream.

## 2.8 Summary

This chapter has described an on-line power estimation and model generation framework. The PowerTutor power estimation tool informs smartphone developers and users of the power consumption implications of decisions about application design and use. The power model in PowerTutor includes seven components: CPU and LCD as well as OLED, GPS, Wi-Fi, audio, and cellular interfaces. For 10-second intervals, it is accurate to within 0.8% on average with at most 2.5% error. PowerTutor is also capable of estimating power consumption of individual application even when there are concurrent applications running. A software implementation of the power estimation tool has been publicly released on the Google Android Application Market. This chapter has also described PowerBooter, an automatic battery state of discharge based power model generation technique. Power-

Booter constructs power model without using a power meter. The result indicates that the power model built with PowerBooter is accurate to within 4.1% of measured values for 10-second intervals.

# CHAPTER III

# Panappticon: Event-based Tracing to Optimize Mobile Application and Platform Performance

## 3.1 Introduction

Most mobile applications are interactive, with a typical user interaction consisting of an input from the user that triggers a series of operations culminating in user-visible output, often an update to the display. User experience directly correlates with the perceived responsiveness [14], so controlling and often minimizing the latency of such transactions is of critical importance to application developers, system designers, and platform manufacturers.

We define a *perceived user transaction* as a series of operations started by the user's manipulation of the device (e.g., a screen touch or key press) and ended by a display update. Intuitively, the transaction captures the interval between the user instructing the device to do something and the expected result being displayed. Despite the importance of such transaction latencies to user experience, there is little existing development tool support to characterize or identify slow transactions, leaving developers in the dark. This is mainly due to the asynchronous, multi-threaded nature of interactive applications. To keep the UI responsive, applications must do lengthy or potentially-blocking operations on background threads, complicating tracking of the execution flow of a single transaction.

Even more difficult, other unrelated applications or system processes running on the same device may influence the perceived performance. Therefore, simply capturing the internal performance of an application is not sufficient to fully characterize transaction latencies. For example, AppInsight [48] is a recent work that instruments application binaries to study user transactions. This approach cannot directly explain poor performance induced by interaction between unrelated processes, such as an application and system process. We show one such example identified by our work in the following paragraph. Indeed, the system must be studied as a whole.

In this work, we describe Panappticon[1], a system that detects perceived user transactions for real-world users and can help diagnose the reasons for poor performance. We illustrate its use by deploying it in a 14-user, one-month user study. Panappticon is useful to three types of people.

- Application developers can use the knowledge of user transactions to find inefficient application code and optimize accordingly. For example, in our study we found that Reddit News, a popular Android application, has many slow transactions due to CPU contention between its thread and non-critical system activities it triggered. Simply delaying the non-critical work until after the user transaction completes would significantly improve the user-perceived performance.

- Operating system designers can use the information to optimize system policies. For example, in our study we found that the default DVFS (Dynamic Voltage and Frequency Scaling) governor included with Android nearly doubles the latency for transactions over 80 ms in duration, despite being designed for interactive workloads. An improved governor is needed for such interactive workloads.

[1] A Panopticon is a type of building that allows a watchman to unobtrusively observe all occupants. http://en.wikipedia.org/wiki/Panopticon

- Hardware designers and manufacturers can leverage the information to make insightful architecture decisions for future devices. For example, we observe that typical non-gaming applications are not parallelized and do not benefit from multi-core processors, suggesting that designers not neglect single-core performance.

Panappticon establishes causal relationships between user inputs and display updates by tracking execution flow between threads, through asynchronous calls, and across interprocess communication boundaries. This is achieved by instrumenting event handlers, asynchronous call interfaces, and the interprocess communication mechanisms in both user space and kernel space to log events from which the execution flow can be reconstructed. The system further logs resource usage information including context switches, network blocks, and disk blocks to help identify the root causes of slow transactions.

We validated Panappticon on ten open-source applications, manually confirming that the detected user transactions and latencies were correct. Panappticon incurs an average 6.1% performance overhead and has unnoticeable impact on battery life.

This work makes three major contributions.

- We describe Panappticon, a system that automatically characterizes perceived user transactions and provides detailed resource usage information, allowing for root-cause diagnosis of the causes of slow transactions. The source code of Panappticon will be made publicly available.

- We provide an unobtrusive methodology for extracting perceived user transaction latencies based on causal relationships in the operations triggered by an input.

- We present results of a real-world user study of 14 Android users running Panappticon on their phones for one month. We present three case studies showing how Panappticon can benefit application developers, system developers, and smartphone manufactures.

The rest of the chapter is organized as follows. Section 3.2 formally describes the

goal of Panappticon. It also gives one illustrative example of how Panappticon works and explains the design challenges we encountered. Section 3.3 describes the methodology used to identify user transactions and the overall system architecture. Section 3.4 describes the detailed implementation for Android and Section 3.5 shows our efforts to validate this implementation and discusses its limitations. Finally, Section 3.6 presents our analysis of the data collected from the user study. This analysis indicates potential application or system performance inefficiencies and suggests possible solutions. Section 5.3 describes related work.

## 3.2 Goals and Design Challenges

We designed and implemented a system that identifies user-perceived transactions for applications and exposes their performance bottlenecks. We anticipate this work to be most useful to application developers, system developers, and smartphone manufacturers who want to improve the user-perceived performance of their designs by reducing transaction latency. In this section, we state our goal, show one illustrative example, and explain the design challenges.

### 3.2.1 Design goal and example

A *perceivable user transaction* refers to a series of operations in the system started by a user's input to the device, e.g., a screen touch or button press, and ending with a display update. The latency between the UI input and the update captures the latency perceived by users. That is, any operation not included in the user-perceived transaction does not influence the perceived latency and therefore does not directly impact user experience. Note that some UI inputs can trigger multiple display updates. In such case, we define the last display update as the end of the transaction.

Figure 3.1 shows one illustrative example of a user-perceived transaction. Imagine

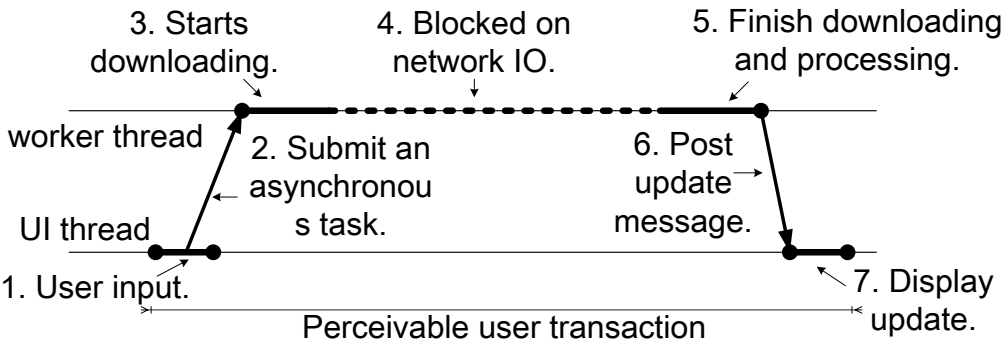



Figure 3.1: Illustrative example of perceivable user transaction: horizontal direction represents time while vertical direction represents different threads.

a simple application that upon a button press, downloads and displays a celebrity quote. As shown in Figure 3.1, the transaction is started by a button press. In the UI event handler triggered by this input, the main UI thread submits an asynchronous task to a worker thread to download the quote. During execution of the asynchronous task, the worker thread may block while waiting for packets from the network. Eventually after the download finishes, the worker thread posts a message back to the UI thread to display the quote. In this example, the operations between the user input and display update form one user-perceived transaction.

Panappticon seeks to identify such user-perceived transactions and identify the performance bottlenecks for each. To achieve this goal in the preceding example, the system must record the user input and, based on the asynchronous call, establish a causal relationship between the UI event handling and the worker thread execution. Then, the system must record the network block and, finally, link the worker thread execution to the UI update by tracking the posted message.

To understand the performance bottleneck for perceived user transaction, we need to identify the critical path for each transaction and understand the dominant component on such path. Critical path is the bottleneck path whose length captures the perceived latency.

Changing the length of any link on such path would also change user perceived latency. In the example above, the path between the input and update through the worker thread represents the critical path and network latency dominates the latency of it.

### 3.2.2 Design challenges

To design a system that achieves the above goal, we are faced with the following challenges:

- *A user transaction can involve execution across thread and process boundaries.* For example, in the preceding example, the main thread submits the real work to be done asynchronously on a background thread. In fact, doing long running operations on background threads is a common Android programming pattern, as explained in Section 3.4. Even more challenging, some applications contain two or more independent processes, one running the UI and the others background work, with all actively involved in a user-perceived transaction. Therefore, we must track asynchronous (and synchronous) calls across thread and process boundaries.

- *There is not perfect temporal correlations between user inputs and display updates.* In the preceding example, it is possible that other display updates are triggered by a change in system state (e.g., a change in network connectivity) between our input and display update. Or in a more complicated application, a second user transaction might start before the first finishes. Both scenarios break the temporal correlations between the user input and display update, ruling out the possibility of simply grouping display updates with the prior input. Consequently, our system must explicitly track causal relationships among operations.

- *It is necessary to know the underlying hardware state.* One of our goals is to help system and application developers determine the reasons for lengthy transactions. Possible

causes include contention for the processor, long blocking times on network or disk IO, or poorly designed system policies such as DVFS. For instance, in our running example, if the network blocking time is reduced, the user-perceived latency would also be reduced. To achieve this goal, we must efficiently log fine-grained resource usage information, such as IO blocking times and context switches.

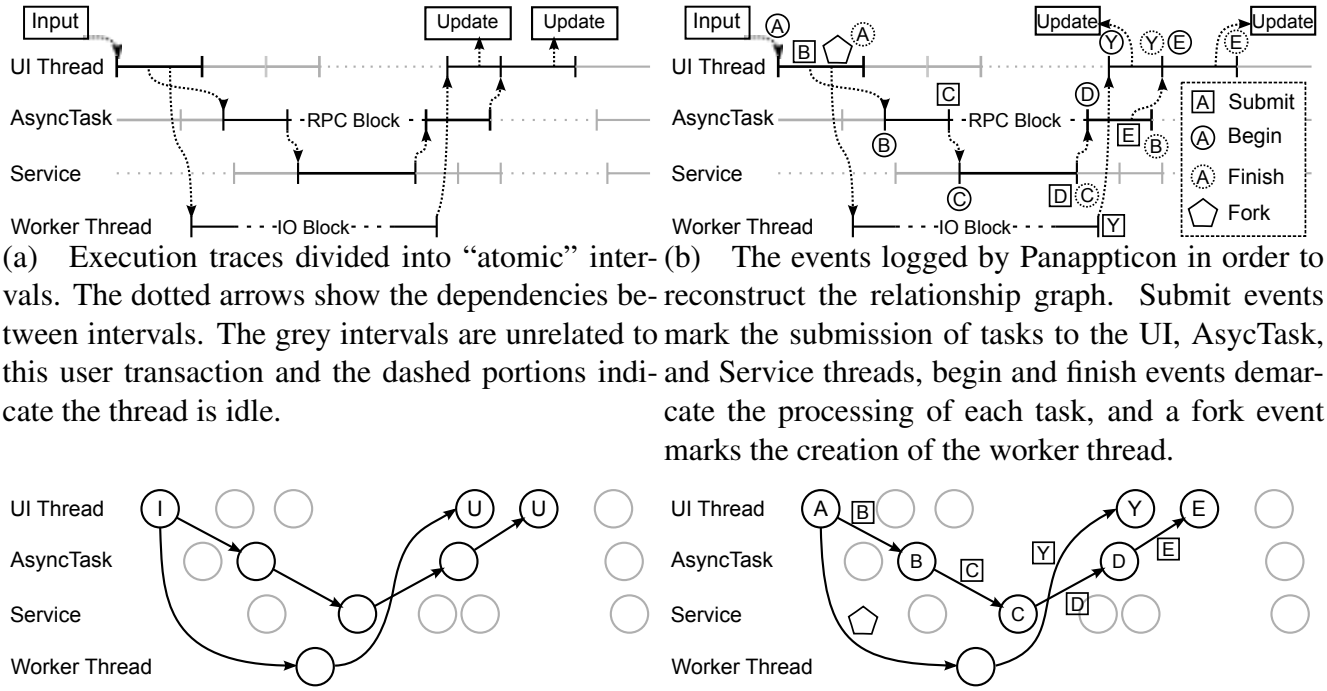
- *Mobile device are resource constrained.* The system must have low performance and energy overheads so as to not influence user experience. Controlling overheads on smartphones with limited CPU throughput, memory, and energy capacity requires efficient logging of only necessary information and relegates the complicated user transaction analysis to the server end.

## 3.3 Approach overview

Motivated by these design challenges, we developed and implemented an event-based tracing methodology that can efficiently capture (1) the relationships between operations to identify user transactions across threads or processes and (2) which resource (e.g., CPU, network, disk, etc.) a thread is using or waiting on at each instant to reveal performance bottlenecks. This section describes our methodology, shown in Figure 3.2.

### 3.3.1 Methodology overview

Tracking user transactions across threads and processes requires separating the execution traces of each thread into "atomic" intervals, identifying the causal relationships among them, and determining the initial and terminal intervals. Such intervals represent work that always happens together, for example, a worker thread processing one task from a task queue. Figure III.2(a) illustrates the execution trace of an example transaction divided into intervals with arrows indicating the causal relationships among them. Fig-




(a) Execution traces divided into "atomic" intervals. The dotted arrows show the dependencies between intervals. The grey intervals are unrelated to this user transaction and the dashed portions indicate the thread is idle.

(b) The events logged by Panappticon in order to reconstruct the relationship graph. Submit events mark the submission of tasks to the UI, AsycTask, and Service threads, begin and finish events demarcate the processing of each task, and a fork event marks the creation of the worker thread.

(c) Relationship graph showing execution dependencies for this transaction. Each node represents one execution interval, with an incoming edge indicating the interval was triggered by the preceding node. Given this graph, the transaction latency is easily computed from the critical path.

(d) From the events logged above, the full relationship graph can be reconstructed. The begin and finish events demarcate the execution boundary for each node and the submit and fork events indicate which node triggered an interval.

Figure 3.2: Example execution sequence illustrating our methodology for user transaction extraction. Figures III.2(a) and III.2(c) illustrate that the sequence can be viewed as a directed acyclic graph of dependent execution intervals. Figures III.2(b) and III.2(d) show how the graph can be reconstructed from a log of simple events. In this transaction, the user input enqueues an AsyncTask, which after communicating with a background service via RPC, updates the display. It also forks a background thread to read from disk and then update the display. The transaction ends after the second display update.

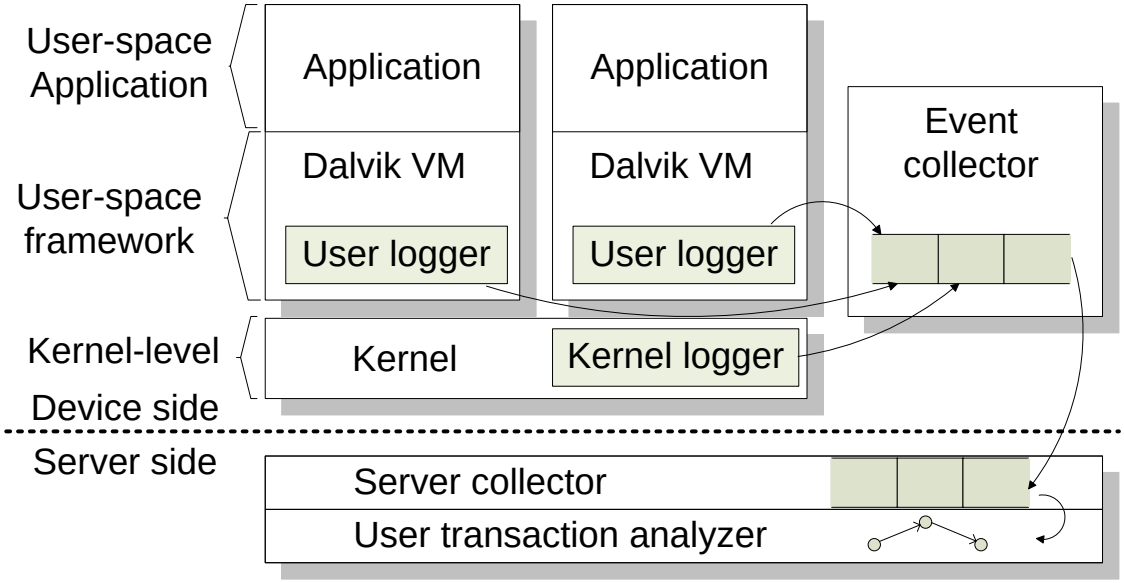



Figure 3.3: System architecture overview.

ure III.2(c) shows the graph abstraction of the causality relationships we wish to obtain.

Thus, to identify user transaction we have three challenges: (1) to split each execution trace into "atomic" intervals, (2) to identify the causal relationships between the intervals, and (3) to identify the intervals representing the beginning (i.e, user input) or end (i.e., display update) of a transaction path. We thus log events sufficient to reconstruct the relationship graphs, as illustrated in Figure III.2(b). For the first and seconds challenges, we identify the popular programming paradigms used in Android and instrument the platform to record events indicating their intervals. For the third, we record user input and display update events in the Android framework code, thus tagging the current interval. From these events, the relationship graph can be reconstructed as shown in Figure III.2(d).

To track which resources a thread is using or waiting on, we record events when each resource is accessed, for example each context switch for the CPU, network blocks, and disk IO blocks. Section 3.4 enumerates all the events we capture.

### 3.3.2 Architecture overview

This section describes the system architecture. Panappticon is composed of the five major components shown in Figure 3.3: a userspace logger, a kernelspace logger, an event collector, a server-side collector and a user transaction analyzer.

**Userspace logger and kernelspace logger:** The userspace and kernelspace loggers record the events mentioned in the preceding section. Specifically, input events, display update events, and most events indicating causal relationships are captured in the userspace logger, where the high-level programming paradigms make event-based inference easier. The kernel logger captures resource utilization events and some events indicating causal relationships across process boundaries, like forking and IPC transactions. Section 3.4 enumerates all the events captured by these two loggers. To minimize the performance impacts of these loggers, both buffer events in memory, sending them to the collector in batch when the buffer is full.

**Event collector and server-side collector:** The event collector is responsible for gathering the traces from the loggers and sending them to server collector for processing. To minimize performance and energy overheads while not losing data, the logs are uploaded in batch on WiFi only. Failed transmissions are buffered to the SDcard and retried later.

**User transaction analyzer:** The user transaction analyzer extracts the relationship graphs pursuant to the methodology of Section 3.3.1. From these graphs, it extracts user transactions and their corresponding user-perceived latencies and resource usage. Section 3.4 details this graph extraction.

## 3.4 Instrumentation details

Panappticon is an instrumented Android system that realizes the user transaction extraction approach described in Section 3.3.1. There remain three major challenges with

Panappticon: (1) what information needs to be captured? Capturing such information should allow us to detect each user transaction and its resource usage with minimum overhead. The answer to this question guides our instrumentation to the system on the device. (2) How do we use these information to construct the relationship graph to link user input with display update and analyze? (3) How can we do resource accounting for each transaction? The answers to the latter two questions describe our approach to conduct analysis on the server side analyzer.

In this section, we first present the background knowledge of Android that is necessary to answer the three questions. Then we will answer these questions separately. In the end, we present a number of example graphs of common Android programming practices.

### 3.4.1 Background: Android

Android is a Linux-based operating system for mobile devices such as smartphones and tablets developed by Google. It differs from a standard Linux distribution due to an additional layer of abstraction between user applications and library, the application framework layer. This layer provides a robust set of programming APIs and a responsive user interface. Android has also been customized to achieve good performance with limited memory and processing power. This subsection summarizes three major differences that are related to our system.

**Dedicated UI handling:** Applications on Android are UI-centric in nature. All UI related events, including handling user interactions and updating the display, are handled on one dedicated thread, which is also the main thread of each application. To keep UI responsiveness, developers should avoid lengthy operations, or blocking on the dedicated UI thread [1]. Instead, they should create a separate worker thread to do most of the work. This is also one of the major reasons that motivates us to track asynchronous calls across

threads in the system.

**Looper thread:** Most Java threads managed by the Android system, including the main thread of each application used for UI events, follow a message queue model. Messages are placed in a queue and the thread loops indefinitely, processing messages from the head of the queue. In Android, such threads are called Looper threads and share a common implementation of the message queue and looper functionality. The shared code facilitates easy logging of message submission and execution events.

**Inter-Process Communication (IPC):** Android makes extensive use of an inter-process communication implementation called Binder to coordinate between system and application processes and to support services, application components that run in the background and may expose methods for remote procedure call (RPC). We instrumented the kernel portion of the Binder implementation to capture such calls.

### 3.4.2 What information do we capture?

There is a trade off between the information our system collects and the overhead of the system. For example, a complete trace of every method in the system framework that applications could potentially call during a transaction would allow us to have full knowledge of the system status. However, such an approach would dramatically slow down the application by orders of magnitude and therefore is not applicable for our purpose. To this end, we record the minimum amount of information that is necessary to determine perceived user transactions and their performance bottlenecks. These information can be categorized into the following types: (1) user interaction events including screen touch and key press, (2) causality between asynchronous calls and callbacks within and across threads, (3) inter-process communication between threads and processes, (4) synchronization mechanisms between threads such as locks and semaphores, (5) resource accounting

Table 3.1: List of Events Captured

| Category | Event type | Space | Data field | Description |
|---|---|---|---|---|
| User inputs | UI_INPUT | User | null | User's input on the touch screen or hardware button |
| | UI_KEY | User | | User manipulation of the software keyboard |
| Async callbacks | ENQUEUE_MSG | User | message_id, queue_id | Enqueues and dequeues a message |
| | DEQUEQUE_MSG | | time_to_dequeue | with message_id on queue with queue_id |
| | SUBMIT_ASYNC_TASK | User | task_id | Submits and consumes a task to one of |
| | CONSUME_ASYNC_TASK | | | the pooled thread executors |
| IPC calls | BINDER_PRODUCE_ONEWAY | Kernel | transaction_id | Caller sends arguments to the remote w/o blocking |
| | BINDER_PRODUCE_TWOWAY | | | Caller sends arguments to the remote and blocks |
| | BINDER_REPLY | | | Remote thread sends return value back to the caller |
| | BINDER_CONSUME | | | A remote thread begins execution |
| Locks: mutex, semaphore, futex, and waitqueue | *type*_WAIT | Kernel | lock_id | Block waiting for access to lock |
| | *type*_WAKE | | lock_id | Resume from block waiting for access to lock |
| | *type*_NOTIFY | | lock_id, notify_pid | Notify waiting thread to wakeup |
| Resource accounting | CONTEXT_SWITCH | Kernel | old_pid, new_pid | Context switch from process old_pid to new_pid |
| | SOCK_BLOCK/RESUME | Kernel | null | Blocks and resumes on socket waiting for connection |
| | DATAGRAM_BLOCK/RESUME | Kernel | null | Blocks and resumes on socket waiting for UDP data |
| | STREAM_BLOCK/RESUME | Kernel | null | Blocks and resumes on socket waiting for TCP data |
| | IO_BLOCK/RESUME | Kernel | null | Blocks and resumes on disk IO |
| Other dependency | FORK | Kernel | parent_pid, child_pid | Parent process forks child process |
| Display update | UI_INVALIDATE | User | null | Invalidates the view, schedules display update |
| | UI_UPDATE | User | null | Redraw the view |
| Additional information | THREAD_NAME | Kernel | t_pid, t_name | Thread t_pid has name t_name |
| | ENTER/EXIT_FOREGROUND | User | null | The current pid enters/exits being foreground app |

for each thread such as context switch, blocking on network and disk IO, (6) other causality relationships between threads, e.g., forking a thread, (7) display update, and (8) additional information that helps to track foreground applications and application names . All the information is summarized as event. A typical event has the following information:

```
Timestamp, Event_type, TID, Data, (CPU_core)
```

Note that `CPU_core` refers to the number of cores on while the event happens. It is only available for kernel events.

Section 3.1 summarizes all the events our system captures along with the information in each event. We are going to explain each type of event now.

**User input:** We record two types of input events: the input from screen touch or hardware button and the input from the software keyboard. In Android, the first type of input is dispatched to the foreground application through the *onInput*() callback method in the View class. We instrumented our hooks in the *onInput*() method. Unlike the first type of inputs, software keyboard inputs are first dispatched to the systemUI application that translates the screen touch event into keyboard inputs. They then get passed to the foreground applications though the input method editor. We instrumented the input method editor in the specific Android library to record software keyboard inputs.

**Asynchronous calls and callbacks:** As we mentioned in Section 4.4.1, the UI centric nature of Android applications requires developers to get length operations done asynchronously though other worker threads. To achieve this, there are two common programming models in Android: (1) starting a worker thread for the work and post a message back to the UI thread to update display after the work is done, or (2) submit a task to the pool of thread executors.

In the first model, we instrumented the MessageQueue class in the Android specific framework library to capture the causal relationship between message enqueue and de-

queue. Note that each message is associated with a unique message ID so that we can easily match the enqueue and dequeue events.

Similarly, in the second model, we instrumented the ThreadPoolExecutor class in the generic Java library to record the causal relationship between task submission and consumption. To match these two events, we record the task ID, which is a hash value calculated from the memory address of the Runnable task.

**Inter-process communication:** As mentioned above, Android uses a kernel-level inter-process communication implementation called Binder for RPC. For each process, Binder manages a pool of threads to execute incoming RPC requests. For each call or Binder *transaction*, we log the full RPC call by tracking events across process boundaries.

**Resource accounting:** To help determine if slow resources or high resource contention caused a slow user transaction, we log access to the three main time-shared resources used by Android applications: processor, network, and disk. For the processors, we log each context switch, including the incoming thread ID, the outgoing thread ID, and the new state of the old thread (still runnable, interruptible sleep, etc.). For the disk and network, we log when a thread blocks on a read request and then when it resumes.

**Synchronization mechanisms:** Contention for virtual resources (worker threads, shared data segments, etc.) could also cause high user transaction latency. Access to such resources is usually mediated by synchronization primitives, so we log contested accesses to the following in-kernel primitives: waitqueues, semaphores, mutexes, and futexes. Specifically, we log when a thread blocks waiting for access, when it resumes from that block, and, after releasing a primitive, which waiting threads are awakened. We do not log lock or unlock events due to the volume of accesses—contested accesses are much rarer. We ignore spinlocks for the same reason.

**Display update:** Android provides developers two major ways to update the display:

the View class in the framework library and OpenGL to render display directly. Our system only focuses on the first way right now because OpenGL is mainly used by complicated graphic rendering in gaming application. The concept of user transaction in those gaming application is different from our definition in interactive applications due to animation. We will discuss this int more detail in Section 3.5.

The View class in Android framework is the basic building blocks for user interaction. Each view occupies a rectangular area and is responsible for drawing and event handling in that area. All the views on the screen are organized as a single tree, where ViewRoot class is the root. Any *invalidate*() in the child view would trigger a view tree traversal to update the display in ViewRoot class. Therefore, we placed our hooks in ViewRoot.

**Additional information:** In addition to all the information necessary to capture causality between events, we also collected additional information to help us better understand the context of the transactions. For example, we recorded the application that enters and exits the foreground to distinguish the application the user interacts with from system applications. This is achieved by modifying the Activity class in the framework. Similarly, the kernel records the name of each thread.

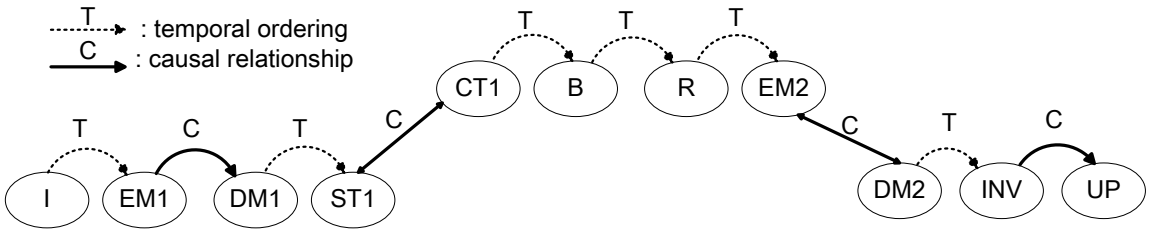



Figure 3.4: Illustrative example of relationship graph.

### 3.4.3 How do we construct relationship graph?

To infer the user transaction from the event stream, we construct directed acyclic graphs based on the relationships between events. Each event entry in the trace is identified as a node in the graph. Each edge between nodes represents the relationship between two events.

Listing III.1: Example trace of correlation graph

```
(I) USER_INPUT     pid:0
(EM1) ENQUEUE_MSG   pid:0     msg_id:1
(DM1) DEQUEUE_MSG   pid:0     msg_id:1
(ST1) SUBMIT_ASYNCTASK pid:0     tk_id:1
(CT1) CONSUME_ASYNCTASK pid:1    tk_id:1
(B) SOCK_BLOCK     pid:1
(R) SOCK_RESUME    pid:1
(EM2) ENQUEUE_MSG pid:1     message_id:2
(DM2) DEQUEUE_MSG pid:0      message_id:2
(INV) UI_INVALIDATE pid:0
(UP) UI_UPDATE     pid:0
```

Figure 3.4 illustrates the relationship graph constructed based on the example trace in Listing III.1. The map between node name and event is also shown in the trace. The logic

of the trace is shown in Section 3.2.

We identified two types of relationships: causal relationship and temporal ordering.

**Causal relationship:** Causal relationship indicates the relationship between two execution intervals where the earlier interval triggers the latter one, as explained in Section 3.3.1. It is represented with edges with solid line in the graph. For example, the message enqueue event triggers the message dequeue action. In other words, without the message enqueue operation, the message dequeue operation would not exist. We identified the following node pairs correlated with causality.

- ENQUEUE_MSG and DEQUEUE_MSG with the same message_id.
- SUBMIT_ASYNCTASK and CONSUME_ASYNCTASK with the same task_id.
- BINDER_PRODUCE or BINDER_REPLY events and BINDER_CONSUME with the same transaction_id.
- Two nodes created by FORK event: the node representing FORK event on the parent thread and the node representing the initial execution on the child thread.
- UI_INVALIDATE and its closest UI_UPDATE within the same thread.

**Temporal ordering:** Temporal ordering between events refers to the relationship between events within an "atomic" execution interval. It is represented with edges with a dotted line in the graph. For instance, in the previous example, message 1 is enqueued while the callback method triggered by the input is executed. Similarly, an asynchronous task is submitted during the execution of the dequeued message 1.

One major challenge with handling temporal ordering is to determine when to end the execution interval. This is important because if not correctly determined, all events on the same thread can connect with each other. As a result, multiple separate user transactions can be mistakenly grouped to one. To overcome this challenge, we categorize the threads in Android applications into two types and we use different approaches for each type as

follows.

- Task-based thread is the most common background thread pattern in Android framework. These threads consume task in a task queue and block when the queue is empty. The end of each task indicates the termination of one execution interval for these threads. For example, the main UI thread in each application is a Looper thread waiting for incoming messages and processing them. All the events happening when processing one message belong to one execution interval. The same approach applies to Binder thread and asynchronous task thread which waits for new transactions and new tasks. Note that for other background threads we don't instrument explicitly, we use events from the locking primitives to indicate the producer/consumer of a task queue and infer execution interval. For example, we did this explicitly with applications using WebKit as explained in Section 3.4.5.

- Another type of thread in Android is worker thread that is forked for one-time execution. These threads exit after the work is done. Therefore, inferring the execution interval is automatically covered by our approach.

Using the methodology above, we can correctly infer separate user transactions when they overlap. By further extracting the critical paths from UI_INPUT and UI_UPDATE in each transaction, we can derive the latency of each perceived user transaction. Note that multiple paths can exist between UI_INPUT and UI_UPDATE. In such cases, we develop a heuristic to pick the critical path, which works as follows. When there are more than one incident edge upon one node, we compare the timestamps of all the events that connect to these edges and pick the path through the latest event as the critical path. As this scenario only occurs in 3.7% of our total transactions, the accuracy of Panappticon depends little on the accuracy of the heuristic.

### 3.4.4 How do we account for resource?

When doing resource accounting for user transactions, the major research question we are trying to answer here is what are the reasons for transactions with noticeable delay? What can the system do to speed them up?

To this end, we analyzed the resource usage on the critical path for each transaction in two steps. First, we add the resource accounting related kernel events into the correlation graph to indicate the usage of certain physical resource. In particular, context switch events represent the thread occupying CPU while blocking on IO or network representing usage on disk IO or network. In additional to physical resource, we also add synchronization events into the graph to represent the consumption of virtual resource. For example, a thread can be blocked on a mutex and wait for other threads to release it. Note that these events are added to the graph based on temporal correlation.

Second, we analyze the edge on the critical path to understand the reason for its latency. All edges can be placed into two major categories as shown below.

- The edge that indicates the corresponding thread is running and occupying the processor. For example, edges between any events that occur between two context switches. Latency due to this type of edge depend on processor speed.

- The edge that suggests the corresponding thread is waiting for some resource and being blocked. We further place different resources in the following categories: (1) physical resource such as network, IO, and CPU. Waiting for CPU means the thread gets context switched out when it is still eligible to run. (2) Virtual resource such as locks or a thread execution. For example, the latency between submission and consumption of an asynchronous task can be because all the worker threads are occupied and therefore blocks the consumption of the new task. Approaches that shorten these edges vary based on specific

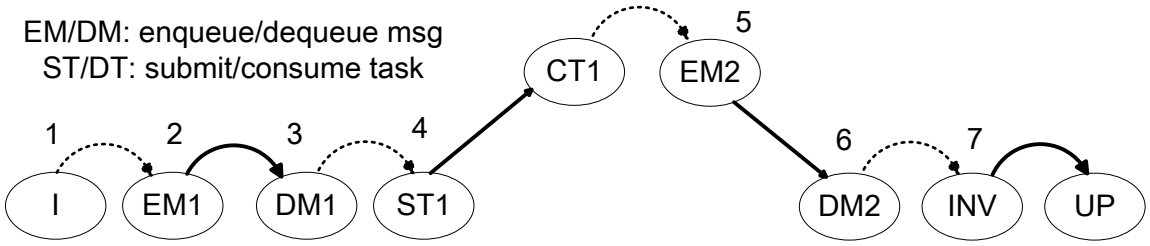



Figure 3.5: Example trace of submitting an asynchronous task to get work done.

resource.

By following the two steps above, we are able to find the time spent on different types of edges for each transaction. We then can infer the reasons for long transaction and propose potential solutions.

### 3.4.5 Example graphs of common programming models

We now present two example relationship graphs generated based on common programming patterns in Android framework. The graph patterns shown in these examples are also the patterns we detected repetitively in the traces gathered from the wild. The first example is the graph generated based on application using asynchronous task. The latter one is generated with application using WebKit to render webpages. Note that our approach is not limited to these two programming patterns.

**AsyncTask:** Using AsyncTask is an effective way to maintain the application's responsiveness. The example trace of such programming model is shown in Figure 3.5. The user transaction starts by (1) a button press, which triggers (2) the button press handler in the application to be enqueued as a message. After (3) the handler gets dequeued, (4) it submits an AsyncTask to an existing worker thread. During the execution of the worker thread, (5) the worker thread posts message containing Runnable to update the display to the main UI's message queue. Eventually (6) the Runnable gets dequeued and executed, which leads to the (7) UI invalidate and UI update.

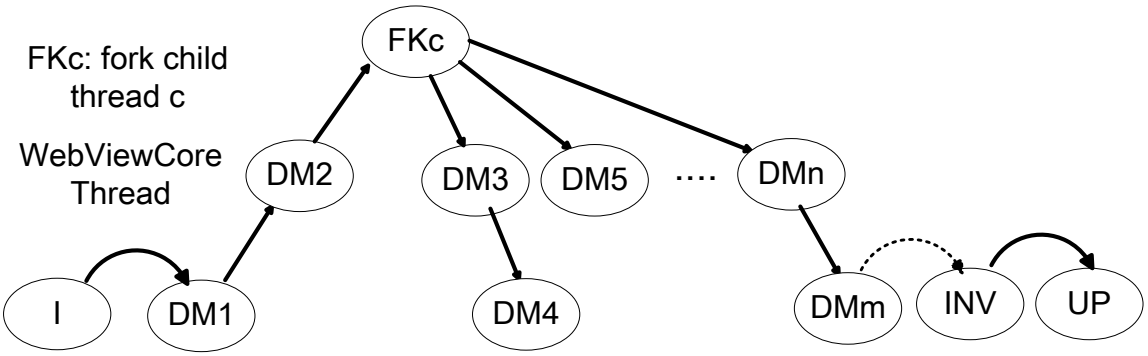



Figure 3.6: Example trace of application using WebView.

**WebView applications:** WebView class is a view that renders web pages using WebKit. It has been extensively used by application developers. Figure 3.6 shows an example of a user transaction that loads a webpage the first time after application launches. Note that for demonstration purpose, all the message enqueue nodes within one execution interval are merged into the prior node. For example, the dequeue of message 1 enqueues message 2 and hence has an outgoing edge to the dequeue of message 2. After the WebViewCore thread dequeues message 2, it (1) forks the rendering thread which prepares different objects on the webpage for rendering. After the preparation, (2) it enqueues a message to the WebViewCore thread, which eventually triggers the display to update.

One major challenge we have for WebView related application is that the rendering thread is a native task-based thread that we do not instrument explicitly. Therefore, to infer the termination of each execution interval, we leverage the kernel waitqueue locking primitives. We observed that when the rendering thread does not have a task to work on, it gets blocked on a waitqueue until some other thread puts a task in the queue and notifies it again. Accordingly, we use the kernel event indicating blocking on the queue to infer the end of the execution interval and use the notify event to infer the causal relationship between producer/consumer threads.

Table 3.2: List of Events Captured

| Type | Name | Description |
|---|---|---|
| Synth app | AsyncApp | Starts an AsyncTask after button press and updates the display |
| | WorkerThreadApp | Forks a worker thread after button press and updates the display |
| | ServiceApp | Starts a remote service, make an IPC call, and updates the display |
| | WebViewApp | Loads a webpage using standard Android API |
| | AnimationApp | Starts an AsyncTask while displays the loading animation, terminates the animation after the task is done |
| Open source app | CrossWords | Loads a cross word game, displays solution based on user inputs |
| | ReadForSpeed | Downloads text and displays it based on timer |
| | Android browser | The default browser on Android |
| | K9 mail | Mail client |
| | NPR news | News reading application |

## 3.5 Validation

We now validate that Panappticon correctly identifies user-perceived transactions and discuss its performance and energy overheads. All experiments are done on Galaxy Nexus phones running Android 4.1.2.

### 3.5.1 Accuracy analysis

We examine the accuracy of Panappticon by evaluating the ten applications, five synthetic benchmarks and five open-source applications, listed in Table 3.2. In these tests, we wish to (1) verify that Panappticon correctly links each UI input to the resulting display updates and (2) confirm that the extracted relationship graphs are correct.

For the first test, we manually instrumented the applications to measure the latency between a user input and the resulting display update. Given the source code, we identify the corresponding method where the application receives and processes users' input and where it renders the view to the display. By comparing the timestamp recorded from these application methods and measured by Panappticon, we concluded that Panappticon correctly identified and linked the inputs and display updates for all ten applications, reporting

the correct transaction latency.

For the second test, we compared visualizations of the generated transaction relationship graphs with our understanding of the source code. For example, we knew from studying the source if an application used an asynchronous task or RPC call to perform part of the transaction. The generated graphs aligned with our expectations, indicating that Panappticon extracted the correct relationship graphs.

Although Panappticon performs well on our intended workloads, it cannot handle all applications and behaviors. We describe some of its limitations here.

- **Approach limitation:** Our approach does not track data or control dependencies directly, but relies on instrumented system or platform libraries that provide support for high-level programming paradigms like task-queues or semaphores. Applications that implement their own coordination primitives or use lockless synchronization cannot be tracked by Panappticon. Tracking data dependencies requires techniques like taint tracking [21] that incur overheads not suitable for online use.

Our definition of user-perceived transaction is not directly suitable for animated applications like games. Our definition tries to capture the time a user spends waiting for an expected result to the input. But with animated activities, the expected result manifests over many frames and may be modified by later inputs. As a result, we exclude animated transactions (like most games) from the experiment, as shown in Section 3.6. Note that according to a recent study, game applications take up roughly 15% of the total number of applications [2].

- **Instrumentation limitation:** Our specific implementation for Android assumes that background worker threads are task-driven threads. Our instrumentation modifies the common Android-provided task-driven primitives like the Looper, AsyncTask, and Executor classes to record the start and end of each task. While this assumption holds true

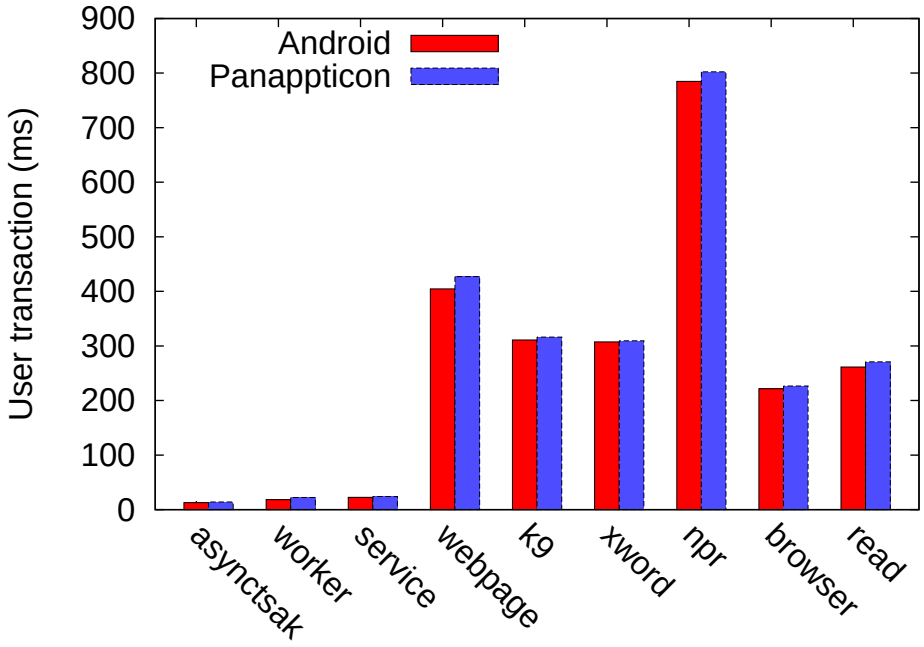



Figure 3.7: Overhead evaluation of Panappticon.

for most interactive applications, it can miss work done on native, non-Java background threads.

One particular example, the WebKit library used to display web pages, is quite prevalent so we use kernel primitives to infer its task intervals as described in Section 3.4. WebKit is the only such library we found in our traces, but other less-frequently used libraries might exist. Panappticon would miss them until the implementation is extended.

### 3.5.2 Overhead analysis

We now present the performance and energy consumption overhead of Panappticon.

**Performance:** The performance overhead of Panappticon stems mainly from the CPU cycles used to log each event and the memory used to store the logs (reducing memory available for the Linux file cache and Android application cache). The CPU overhead is minimized by eliminating locking in the logging path (e.g., by using per-CPU log buffers in the kernel) and memory overhead is minimized by fixing the buffer size to 30 MB in the kernel and 15 MB in userspace.

To evaluate the overhead, we compared user transaction latencies on a system with Panappticon to one without. To get latencies on the system without, we manually instrumented several open source applications to directly record transaction lengths. The experiment was conducted on two Galaxy Nexus phones. Shown in Figure 3.7, the average overhead of Panappticon is 6.1%.

**Battery:** The energy consumption of Panappticon is mainly due to uploading the logs to our server. This cost is reduced by transferring data only when WiFi is available. When not available, Panappticon saves the data to flash and defers the transmission. The saved data is not sent until the phone is charging and WiFi is available. No users reported a noticeable change in battery life.

## 3.6 Case studies from real-world traces

We now present the result from our deployment of Panappticon in the wild. We first explain the setup of the deployment and then proceed through three case studies on the real-world traces, showing three specific findings uncovered by Panappticon that would interest application developers, system designers, and smartphone manufacturers.

### 3.6.1 Experimental setup

We recruited 14 students from University of Michigan as volunteer users to deploy Panappticon. We selected users based on one major criteria—that they are regular, long-time smartphone users—to ensure that their user behavior is representative of smartphone users. All users used Galaxy Nexus phones [5], which has a 1.2 GHz dual-core processor.

The deployment has two major goals. (1) Identify the bottlenecks responsible for transactions of noticeable length (i.e., >50–100 ms). This goal can be achieved by looking at the resource usage for each transaction. (2) Understand the impact of architectural differences on user transactions, e.g., multi-core devices compared with single-core device and

Table 3.3: Deployment Statistics

| | |
|---|---|
| Total num of transactions | 104,588 |
| Transactions without animation | 88,656 |
| Animation transactions | 15,932 |
| Application count | 189 |
| Start time | Oct. 31th |
| End time | Nov. 30th |

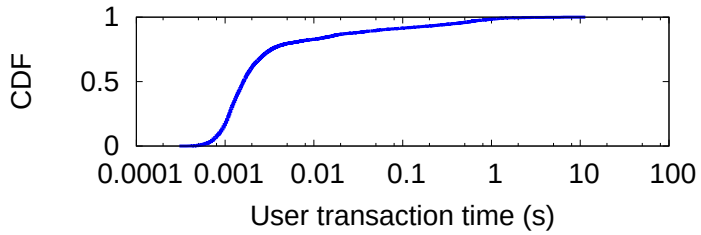
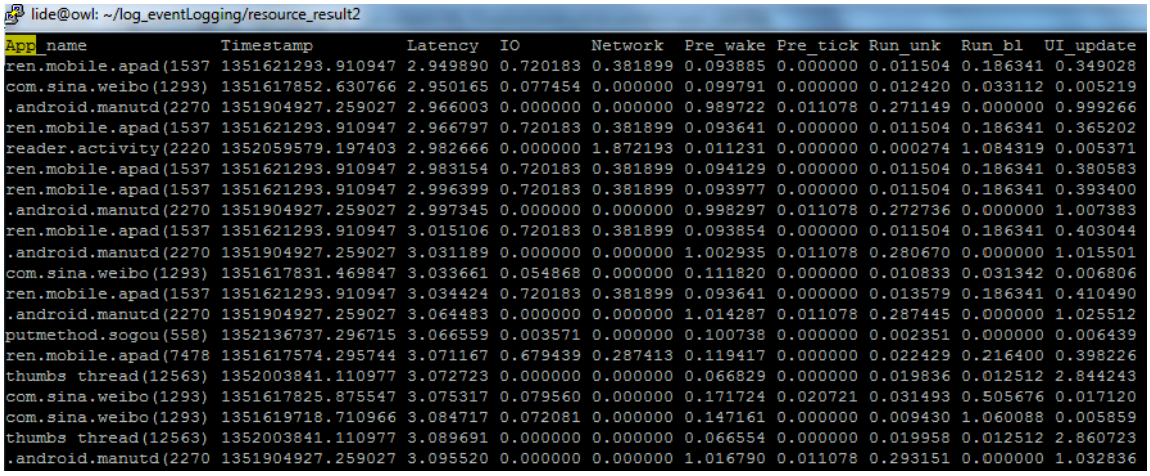



Figure 3.8: Distribution of all non-animation transactions.

the impact of the DVFS (Dynamic Voltage and Frequency Scaling) policy. To achieve this, we periodically disable one core on the device and toggle the DVFS settings. Specifically, we run a daemon that changes the number of cores and DVFS policy every 10 minutes, randomly switching between the four possible configurations. To prevent the transition from interfering with user experience when intensive workload is running, the switching transition only happens when the system is not busy (i.e., with CPU utilization less than 5%).

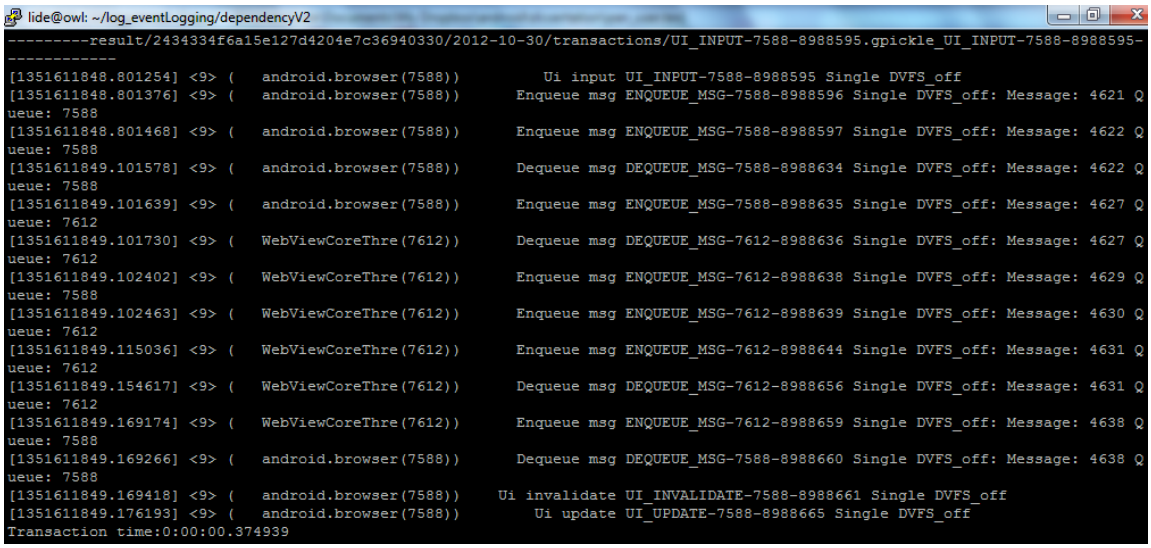
Table 3.3 summarizes the basic statistics of the deployment. We detected 104,588 transactions in total. Among them, 88,656 transactions do not have involve animation. Figure 3.8 presents the cumulative distribution of latencies for all user transactions. Transactions without animation last at most 38.60 seconds with only 2% of transactions lasting longer than 1 second. Note that as explained in Section 3.5, we only focus on non-animation transactions in the next subsection because the length of animation transactions does not directly correlate with user experience.



lide@owl: ~/log_eventLogging/resource_result2

| App_name | Timestamp | Latency | IO | Network | Pre_wake | Pre_tick | Run_unk | Run_bl | UI_update |
|---|---|---|---|---|---|---|---|---|---|
| ren.mobile.apad(1537 | 1351621293.910947 | 2.949890 | 0.720183 | 0.381899 | 0.093885 | 0.000000 | 0.011504 | 0.186341 | 0.349028 |
| com.sina.weibo(1293) | 1351617852.630766 | 2.950165 | 0.077454 | 0.000000 | 0.099791 | 0.000000 | 0.012420 | 0.033112 | 0.005219 |
| .android.manutd(2270 | 1351904927.259027 | 2.966003 | 0.000000 | 0.000000 | 0.989722 | 0.011078 | 0.271149 | 0.000000 | 0.999266 |
| ren.mobile.apad(1537 | 1351621293.910947 | 2.966797 | 0.720183 | 0.381899 | 0.093641 | 0.000000 | 0.011504 | 0.186341 | 0.365202 |
| reader.activity(2220 | 1352059579.197403 | 2.982666 | 0.000000 | 1.872193 | 0.011231 | 0.000000 | 0.000274 | 1.084319 | 0.005371 |
| ren.mobile.apad(1537 | 1351621293.910947 | 2.983154 | 0.720183 | 0.381899 | 0.094129 | 0.000000 | 0.011504 | 0.186341 | 0.380583 |
| ren.mobile.apad(1537 | 1351621293.910947 | 2.996399 | 0.720183 | 0.381899 | 0.093977 | 0.000000 | 0.011504 | 0.186341 | 0.393400 |
| .android.manutd(2270 | 1351904927.259027 | 2.997345 | 0.000000 | 0.000000 | 0.998297 | 0.011078 | 0.272736 | 0.000000 | 1.007383 |
| ren.mobile.apad(1537 | 1351621293.910947 | 3.015106 | 0.720183 | 0.381899 | 0.093854 | 0.000000 | 0.011504 | 0.186341 | 0.403044 |
| .android.manutd(2270 | 1351904927.259027 | 3.031189 | 0.000000 | 0.000000 | 1.002935 | 0.011078 | 0.280670 | 0.000000 | 1.015501 |
| com.sina.weibo(1293) | 1351617831.469847 | 3.033661 | 0.054868 | 0.000000 | 0.111820 | 0.000000 | 0.010833 | 0.031342 | 0.006806 |
| ren.mobile.apad(1537 | 1351621293.910947 | 3.034424 | 0.720183 | 0.381899 | 0.093641 | 0.000000 | 0.013579 | 0.186341 | 0.410490 |
| .android.manutd(2270 | 1351904927.259027 | 3.064483 | 0.000000 | 0.000000 | 1.014287 | 0.011078 | 0.287445 | 0.000000 | 1.025512 |
| putmethod.sogou(558) | 1352136737.296715 | 3.066559 | 0.003571 | 0.000000 | 0.100738 | 0.000000 | 0.002351 | 0.000000 | 0.006439 |
| ren.mobile.apad(7478 | 1351617574.295744 | 3.071167 | 0.679439 | 0.287413 | 0.119417 | 0.000000 | 0.022429 | 0.216400 | 0.398226 |
| thumbs thread(12563) | 1352003841.110977 | 3.072723 | 0.000000 | 0.000000 | 0.066829 | 0.000000 | 0.019836 | 0.012512 | 2.844243 |
| com.sina.weibo(1293) | 1351617825.875547 | 3.075317 | 0.079560 | 0.000000 | 0.171724 | 0.020721 | 0.031493 | 0.505676 | 0.017120 |
| com.sina.weibo(1293) | 1351619718.710966 | 3.084717 | 0.072081 | 0.000000 | 0.147161 | 0.000000 | 0.009430 | 1.060088 | 0.005859 |
| thumbs thread(12563) | 1352003841.110977 | 3.089691 | 0.000000 | 0.000000 | 0.066554 | 0.000000 | 0.019958 | 0.012512 | 2.860723 |
| .android.manutd(2270 | 1351904927.259027 | 3.095520 | 0.000000 | 0.000000 | 1.016790 | 0.011078 | 0.293151 | 0.000000 | 1.032836 |

Figure 3.9: Perceived user transaction list detected by Panappticon.



lide@owl: ~/log_eventLogging/dependencyV2

```
---------result/2434334f6a15e127d4204e7c36940330/2012-10-30/transactions/UI_INPUT-7588-8988595.gpickle_UI_INPUT-7588-8988595-
------------
[1351611848.801254] <9> (   android.browser(7588))         Ui input UI_INPUT-7588-8988595 Single DVFS_off
[1351611848.801376] <9> (   android.browser(7588))      Enqueue msg ENQUEUE_MSG-7588-8988596 Single DVFS_off: Message: 4621 Q
ueue: 7588
[1351611848.801468] <9> (   android.browser(7588))      Enqueue msg ENQUEUE_MSG-7588-8988597 Single DVFS_off: Message: 4622 Q
ueue: 7588
[1351611849.101578] <9> (   android.browser(7588))      Dequeue msg DEQUEUE_MSG-7588-8988634 Single DVFS_off: Message: 4622 Q
ueue: 7588
[1351611849.101639] <9> (   android.browser(7588))      Enqueue msg ENQUEUE_MSG-7588-8988635 Single DVFS_off: Message: 4627 Q
ueue: 7612
[1351611849.101730] <9> (   WebViewCoreThre(7612))      Dequeue msg DEQUEUE_MSG-7612-8988636 Single DVFS_off: Message: 4627 Q
ueue: 7612
[1351611849.102402] <9> (   WebViewCoreThre(7612))      Enqueue msg ENQUEUE_MSG-7612-8988638 Single DVFS_off: Message: 4629 Q
ueue: 7588
[1351611849.102463] <9> (   WebViewCoreThre(7612))      Enqueue msg ENQUEUE_MSG-7612-8988639 Single DVFS_off: Message: 4630 Q
ueue: 7612
[1351611849.115036] <9> (   WebViewCoreThre(7612))      Enqueue msg ENQUEUE_MSG-7612-8988644 Single DVFS_off: Message: 4631 Q
ueue: 7612
[1351611849.154617] <9> (   WebViewCoreThre(7612))      Dequeue msg DEQUEUE_MSG-7612-8988656 Single DVFS_off: Message: 4631 Q
ueue: 7612
[1351611849.169174] <9> (   WebViewCoreThre(7612))      Enqueue msg ENQUEUE_MSG-7612-8988659 Single DVFS_off: Message: 4638 Q
ueue: 7588
[1351611849.169266] <9> (   android.browser(7588))      Dequeue msg DEQUEUE_MSG-7588-8988660 Single DVFS_off: Message: 4638 Q
ueue: 7588
[1351611849.169418] <9> (   android.browser(7588))    Ui invalidate UI_INVALIDATE-7588-8988661 Single DVFS_off
[1351611849.176193] <9> (   android.browser(7588))       Ui update UI_UPDATE-7588-8988665 Single DVFS_off
Transaction time:0:00:00.374939
```

Figure 3.10: Example trace of events on critical path for one transaction.

### 3.6.2 How to use Panappticon to understand performance inefficiency

Panappticon intends to help developers to identify performance inefficiencies in mobile system and applications. To achieve this, Panappticon provides the following information to developers, guiding them to understand the causes of the latency for each user transaction.

- User transaction list sorted by latency. The latency is also broken down by different causes. Figure 3.9 shows an example of the list. It makes it easy for developers

to understand which user transaction should be focused on, e.g., the frequent transactions with noticeable latency, and also what reason contributes most to the transaction latency.

- Event list on critical paths for each transaction. Figure 3.10 shows the example critical paths for one transaction. This piece of information helps developers to know what happens within each transaction in great detail, allowing them to better understand what is the bottleneck for performance inefficiency.

- All events happens during each transaction. This information is also useful as sometimes events that are not on critical paths can also influence the performance of one transaction via resource contention. The format of this event list is the same as the event list of critical paths as shown above.

With these information provided, developers could first identify frequent transactions with noticeable latency, and then explore the dominant reason for its latency by looking at the event list on critical paths. For example, to identify inefficient applications, we first looked at the sorted user transaction list and observed that the Reddit News application had many frequent transactions with noticeable latency. We then extracted the critical path information associated with these transactions and found the events that took a significant percentage of the total transaction latency. The following subsection discusses our findings for this application.

### 3.6.3 Case Study One: Analysis of long transactions for applications

One major goal of Panappticon is to help application developers identify user transactions that may be noticeably and annoyingly long and help expose potential fixes. We now describe how Panappticon identified a performance inefficiency in Reddit News, a popular application for browsing the website reddit.com.

Reddit News is a popular closed-source application on the Android Market that has

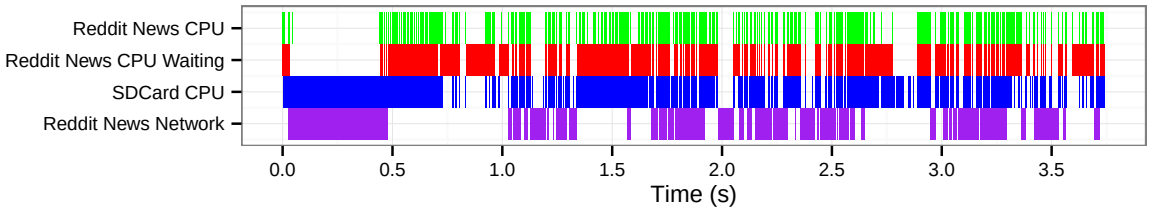



Figure 3.11: (Color) Transaction trace for Reddit News showing the main thread contending for the CPU with system-owned threads used to control the emulated SD card. For the Reddit News thread, time spent using the CPU is shown in green and time waiting for the CPU is in red. Waiting until after the display is updated to cache the downloaded data to the SD card would reduce this contention and shorten the user-perceived transaction latency.

Table 3.4: Resource Accounting Statistics for Sample Transactions from Reddit News

| Total latency(s) | Network block(s) | IO block(s) | Waiting for CPU(s) |
|---|---|---|---|
| 3.78 | 0.98 | 0 | 1.39 |
| 2.35 | 0.42 | 0.02 | 0.93 |
| 1.54 | 0.23 | 0 | 0.89 |
| 1.27 | 0.15 | 0 | 0.33 |

millions of downloads. When doing transaction analysis, Panappticon revealed a large number of transactions with latencies longer then 1 second. Table 3.4 shows the resource accounting statistics generated from Panappticon for four example transactions. "Network" and "IO" block columns show the time spent blocked by those resources on the critical path. "Waiting for CPU" shows the time spent waiting for the CPU while preempted. The rest of the transaction is spent running on CPU.

The tables shows the time spent in preemption is the dominant reason for the high latency and suggests heavy CPU contention during the transactions. Interestingly, a deeper look at the critical paths of these transactions reveals two things: (1) the preempting threads that consume the most CPU time during these transactions are the system threads responsible for writing to the emulated SD card. (2) The preemption is triggered by the Reddit News thread after each network block. Longer network blocks trigger longer preemption times. Figure 3.11 shows one such transaction trace and the CPU usage from the

contending system thread. We believe the Reddit News thread is fetching images from the network and caching them to disk right after download.

Although saving to the SD card does not block the thread directly via disk IO block, the system thread in charge of SD card writing creates heavy CPU contention and blocks the thread on the critical path. Now the remaining question is why the thread writing to SD card starts working during a user transaction. A deeper investigation into the writing policy reveals that although the write back activity happens asynchronously with the write system call, it gets triggered everytime the write buffer gets full. In the Reddit News scenario, each image ranges from 15KB to 3MB while the default buffer size is 8KB. This means that every time an image is downloaded, the SD card thread would be triggered to write it back to SD card and as a result creates the resource contention with the critical path. As we don't have access to the source code of SD card driver, we hypothesize the intensive CPU load being the deficiencies in the driver resulting from wear leveling, the technique that is used to remap blocks to spread write between sectors to reduce SD card wear out. This performance inefficiency induced by SD card is also reported similarly by [35].

There are two potential solutions that can resolve such contention and reduce user-perceived latency. The first solution is to fix the inefficiency in the driver code. This solution fixes the fundamental problem, however, it is beyond many developers. The second solution is that the application developers should defer the image caching after it being displayed to the user or use a larger buffer size.

In addition to the preemption latency, latency due to network also contributes to the long transactions. This suggests that the developers should consider a prefetching or caching policy depending on the specific usage scenario.

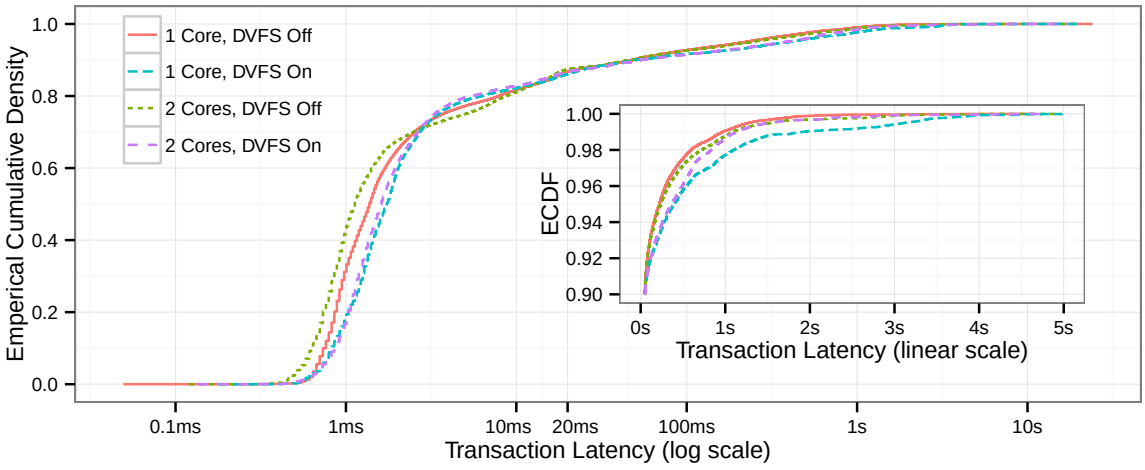



Figure 3.12: Distributions of user transaction latency for different core count and DVFS configurations. The zoomed-in inset plot highlights the differences in the upper percentiles.

### 3.6.4 Case Study Two: Impact of DVFS policy on user transaction latency

The second case study illustrates how Panappticon can help system designers by studying the impact on user transaction length of a specific system setting, the dynamic voltage and frequency scaling (DVFS) policy. DVFS is used to improve energy efficiency by decreasing the processor voltage and frequency to the most energy-efficient point that still provides the needed throughput. This strategy is a problem for latency-sensitive tasks that are often low-throughput and thus do not trigger the faster processor states. Android ships with a DVFS policy, *interactive*, that attempts to improve performance for latency-sensitive, interactive workloads, but as we show in this section, only partially succeeds.

Figure 3.12 shows the empirical distribution of user transaction latencies for four different CPU configurations, DVFS on (using the *iteractive* governor) and off (using the highest frequency, 1.2 GHz) with both one and two cores available. We observe that for both dual and single core configurations, user transaction latencies are higher with DVFS enabled. For short transactions below 20 ms or so, the difference is probably unnoticeable, but the differences are significant at higher latencies. For example, the difference at the

96th percentile is 170 ms for the dual core configurations and 517 ms at the 98th percentile for the single core configurations. These observations indicate that DVFS negatively and noticeably impacts user transaction latency.

To understand why this behavior occurs, we examine the details of the DVFS policy. The *iteractive* governor uses a standard windowed-utilization policy to set the CPU state—the frequency is matched to the CPU utilization over the preceding 20 ms—but includes three optimizations for latency-sensitive tasks. First, the CPU speed is boosted (to 700 MHz) on each user input (touchscreen or keyboard). Second, the CPU speed is boosted (to 700 MHz) if the CPU utilization exceeds some threshhold (85%) over the short duration (20 ms) after leaving idle, bypassing any intermediate states. Third, the CPU is held at a frequency for a minimum duration (60 ms) before it may be lowered (it may increased with no delay). From this explanation, we hypothesize that short transactions are slower because the CPU speed is initially boosted to only 700 MHz, not the full 1.2 GHz, and longer transactions are slower because the CPU speed is allowed to drop after 60 ms.

A QQ plot comparing the latency distributions for a single core configuration (Figure 3.13) supports this hypothesis. Below 60 ms, the distributions are essentially the same, with DVFS off being slightly faster below 10 ms. Above 60 ms however, DVFS is much worse. Latencies average 1.75× higher for transactions above 100 ms. As illustrated in Figure 3.14, most long transactions include network and disk blocking interleaved with the CPU usage and thus despite using the CPU for relatively long (e.g., 1 second), do not exhibit high CPU utilization. After 60 ms, *interactive* allows the CPU speed to drop to match the utilization and the remaining processing is done at a lower speed, increasing the transaction latency. For interactive workloads with network and disk blocking, CPU utilization is a poor measure for controlling the frequency.

Given the preceding observations, we offer two suggestions for improving the DVFS

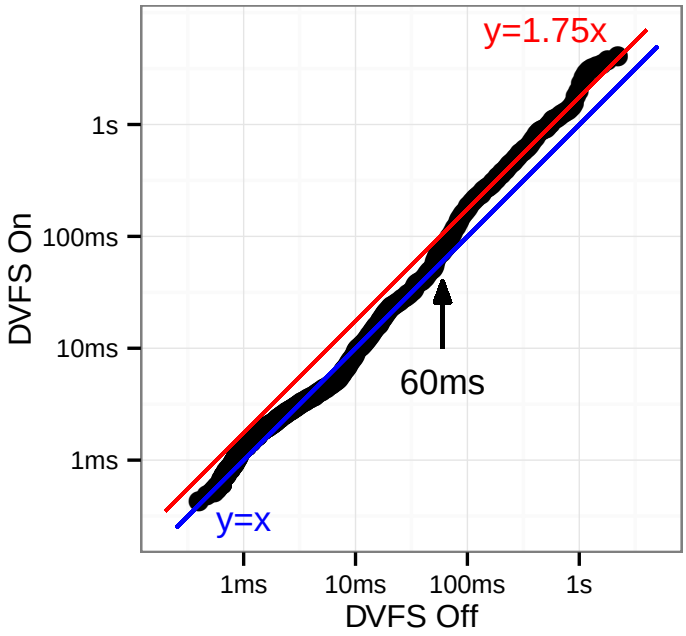



Figure 3.13: QQ plot comparing latency distributions with DVFS on and off for a single core. For transactions with latencies below 60 ms, DVFS has little impact, but for longer transactions, DVFS hurts latency by as much as 1.75×. The break occurs at 60–80 ms because the *interactive* governor allows the frequency to drop 60 ms after the user input.

policy for interactive workloads. The first is simple—increase the minimum duration after user input before allowing the frequency to drop. This requires no changes to code (the duration is already configurable) but would have some unnecessary impact on energy consumption, since the majority of transactions are shorter than the current 60 ms threshold. The second, while more sophisticated, is only a high-level suggestion. Treat the time spent blocking on network or disk as time on the CPU. This renders the CPU utilization (and thus frequency) independent of the time spent blocking, keeping the frequency high when the task is in-progress and allowing it to drop on task completion. Of course, the devil is in the details (e.g., false positives due to web socket threads blocked indefinitely waiting for updates from a server) and since the focus of this chapter is using Panappticon to illuminate problems, not develop new DVFS policies, we leave further evaluation to future work.

Table 3.5: Power and Energy Consumption for Different Frequency Levels for Galaxy Nexus

| Frequency | CPU Power (mW) | Total Power (mW) | Estimated energy |
|---|---|---|---|
| 350 | 220 | 820 | 2.64 |
| 700 | 610 | 1260 | 1.46 |
| 920 | 1000 | 1650 | 1.16 |
| 1200 | 1600 | 2260 | 1 |

Some may argue that DVFS policy intends to trade performance for energy consumption. Therefore, the energy benefit can overweigh performance degradation induced by DVFS policy. However, our measurement result shows when considering the whole system's energy consumption, improving the performance of the system also leads to better energy efficiency due to the energy consumed by system peripherals other than the processor. Table 3.5 shows the measured power consumption of Galaxy Nexus on different frequency levels. Note that these data are gathered using a power meter attaching to the phone when it runs synthetic workload keeping the CPU 100% utilized with different frequency levels. To understand the effect of DVFS policy, take CPU-bounded workload for example. The performance of such task is inversely correlated with the frequency of the processor. When considering the CPU energy consumption alone, the energy efficiency increases with decreasing frequency level. However, when we consider energy consumption of the whole platform, the energy efficiency decreases with decreasing frequency level. Table 3.5 indicates that enabling DVFS policy does not necessarily lead to better energy efficiency. Although it benefits the total energy consumption when the workload is IO-bounded or memory-bounded, when the workload is CPU-bounded as most gaming applications are, DVFS policy would hurt the total energy consumption.

### 3.6.5 Case Study Three: Impact of hardware resource on user transactions

The third and final case study shows how Panappticon can be used by hardware and platform designers to study the impacts of hardware design choices on user transactions.

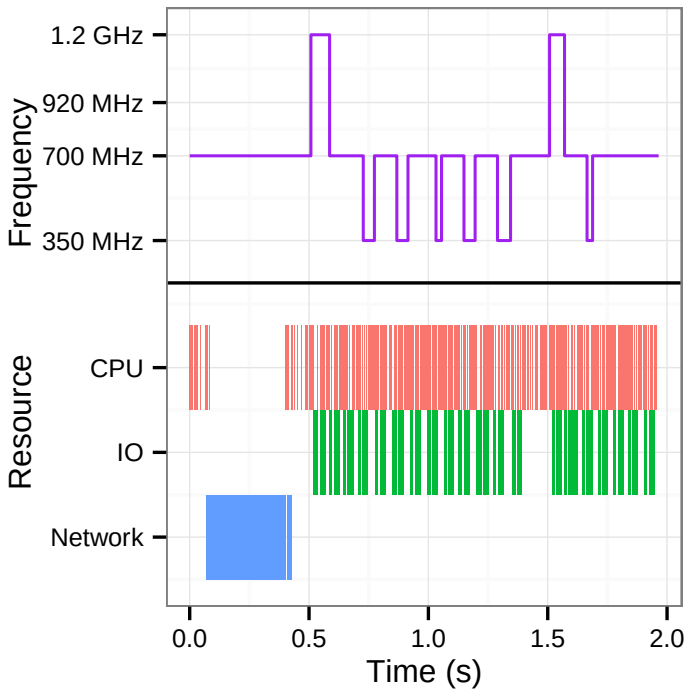



Figure 3.14: transaction illustrating the reason for the poor behavior of the DVFS policy. The CPU use is interleaved with disk blocks and thus although the transaction includes significant CPU time, the utilization is low and the DVFS policy keeps the CPU frequency well below the max.

To this end, we explore one major question: how do multicore processor influence the latency of user transactions?

**How are additional cores used?**

To answer the question, we compare the distribution of user transaction latencies for single core and dual core configurations. To eliminate the effect of DVFS, we consider only the configurations with DVFS disabled. As shown in Figure 3.12, for short transactions below 2 ms, an additional core reduces transaction time. However, for longer transactions, the additional core does not significantly change transaction time, on average. This suggests that longer transactions are not parallelized, CPU-bound workloads[2].

This finding indicates that developers are not writing parallel (non-gaming) applica-

[2]N.B., we do not consider applications using OpenGL, like games. The multicore behavior for such applications may be different.

tions, even when the transactions latencies are high enough to justify it. Although the applications use multiple threads, usually only a single thread is active. Determining if the applications are parallelizable is beyond the scope of Panappticon, but we can offer two suggestions.

First, Panappticon can identify applications with slow, CPU-bound, and non-parallelized execution. Developers of these applications should consider trying to parallelize them. With Panappticon, it is easy to focus on the sections that would provide the most benefit if parallelized.

Second, hardware manufacturers should continue to optimize for single-threaded execution. Many are already doing this, for example, allowing additional cores to be turned off when not in used, providing a single core that operates at a slower, more energy-efficient speed, or providing a single core that operates at higher speed when the others are turned off. Panappticon shows that these features are still suitable for many real user transactions and could be used to quantify their impact.

## 3.7 Summary

In this chapter, we described Panappticon, a system that records events generated in operating system and framework libraries, correlates related events and extracts individual perceived user transaction. Panappticon is able to identify the duration and the critical path of each transaction and disclose performance bottlenecks in each transaction.

This chapter shows that by providing the user transaction information, Panappticon can benefit application developers, system developers, and smartphone manufactures. Panappticon can help application developers identify application design inefficiency, even when the root cause is subtle. Panappticon can help system developers understand the impact of system policy on user transactions, e.g., DVFS policy. This knowledge would enable them

to better optimize their designs. In the end, Panappticon also helps smartphone manufactures to understand the impact of architectural decisions on user transactions, e.g., impact of additional core. This information can guide future design decisions.

# CHAPTER IV

# ADEL: Automatic Detector of Energy Leaks for Smartphone Applications

## 4.1 Introduction

Energy is a scarce resource for smartphone users. Available energy is constrained by limits on battery size and weight, and improvements in battery technology have historically been slow. Moreover, energy demands often increase with the addition of new hardware and software features. To make matters worse, operating systems and applications frequently consume energy to perform tasks that are ultimately useless, a phenomenon we call *energy leaks*. Smartphone users commonly complain about the effects of energy leaks. For example, many iPhone users reported a sudden drop in battery life from 100 hours standby to 6 hours standby due to new energy leaks in Apple's iOS 5.

Energy leaks are in general difficult to detect and isolate [43]. There are two main reasons for this difficulty. First, it is much more difficult to determine how much energy an application leaks than to determine how much energy it uses. Some applications necessarily consume a lot of energy while wasting little (e.g., YouTube). Second, there can be a wide range of causes for unintended energy use, e.g., incorrect API use or poor application design. As we will demonstrate in Section 4.6, even mature and carefully developed operating systems or applications can contain energy leaks.

Energy leaks are tasks that consume energy without ever directly or indirectly influencing the user-observable output of the smartphone. More specifically, if removing a task from the application never directly or indirectly changes any output presented to users, the task is unnecessary, and the energy consumed by it is wasted. We consider energy leaks resulting from two sources. The first source is unambiguous programming errors, e.g., frequently accessing sensor data without using it. The second source is application designs that rely on predictions that are too inaccurate for their intended purposes, as can happen with applications that use overly aggressive prefetching. In addition to identifying unambiguous energy bugs, our work can also determine the percent of energy wasted as a result of prefetching misprediction and identify the particular prefetched data elements that are not ultimately used, thereby supporting improvements to predictors or adaptation of prefetching aggressiveness.

We focus here on detecting and isolating energy leaks in network communication for mobile applications. We choose to focus on network communication because network-intensive applications are very common in smartphones, network communication, including both Wi-Fi and 3G interace, accounts for 39% of system-wide energy consumption, and (most importantly) developers have a hard time optimizing for network energy efficiency, resulting in substantial energy waste [45]. *More specifically, we provide a tool that detects useless network packets which directly leads to energy leak.*

We describe the design, implementation, and evaluation of ADEL (Automatic Detector of Energy Leaks). ADEL is an extension of the Android platform that tracks the information flow of network traffic through applications. By tracking all the computations that depend on each network packet, ADEL is able to determine whether or not a packet constitutes an energy leak, e.g., whether it is deleted before directly or indirectly influencing outputs. ADEL uses dynamic taint-tracking analysis to detect energy leaks. It

automatically labels each data object with a tag when it is first downloaded, then follows and propagates the tag when new data objects are derived from the tainted object. Thus, system outputs can be associated with the downloaded data that influenced them. ADEL provides developers with insight on the use of each packet, allowing design flaws resulting in energy waste to be easily detected and isolated.

We evaluated the effectiveness of ADEL by profiling five open source applications and three closed source applications. For the open source applications, we compared the ADEL-detected wasted network communication with our understanding of logic in the source code. For the closed source applications, we manually correlated the detected tainted object with the content being displayed.

Our ADEL-supported analysis of 15 applications revealed four categories of energy leaks:

1. Misinterpretation of callback APIs that results in wasteful downloads. For example, an application containing a separate downloading thread might not kill the thread properly, allowing the download to continue after the application exits the foreground.

2. Inefficient data refreshing behavior that ignores the application and device status. For example, a widget might be updated whether or not the display is currently on.

3. Repetitive downloads. For example, an application might download repetitive content because it fails to cache downloaded content.

4. Overly aggressive prefetching.

Eliminating or reducing energy leaks of these kinds in our test suite reduces the network energy consumption by approximately 50%. In summary, ADEL effectively helps identify and isolate communication energy leaks.

This work makes the following contributions.

- We provide a definition of *energy leaks*: the energy consumed in operations that

never influences any output produced by a system.

- We describe ADEL, a system-wide infrastructure that uses dynamic taint-tracking to identify network energy leaks. ADEL tracks data originating at the network interface through applications to its eventual use or deletion. Automatically determining whether particular network data are ever used can help applications developers and users to identify, isolate, and fix energy leaks. ADEL is the first system that uses taint-tracking for energy efficiency analysis.

- We used ADEL to profile 15 applications. Six of these have energy leaks that account for more than 40% of their communication energy use. By further examining these leaky applications, we identified four common causes. Our analysis may be useful for both application and platform developers.

The rest of this chapter is organized as follows. Section 4.2 formally defines *energy leak* and defines the problem ADEL seeks to solve. Section 4.3 describes an illustrative example of ADEL's use and explains its design challenges. Section 4.4 describes the implementation of ADEL in detail. Section 4.5 explains how ADEL is validated and points out its limitations. Section 4.6 describes our analysis of energy leaks in 15 applications and indicates major classes of design characteristics resulting in energy leaks. Section 4.7 concludes the chapter.

## 4.2 Problem Definition

We define an *energy leak* as *energy consumed that does not ever influence the outputs of a computer system, e.g., through display interface, audio interface or network interface*. In other words, any energy consumed by operations that never change system outputs, directly or indirectly, is wasted. Eliminating such operations cannot cause observable changes in application behavior. For example, the widget of a news application might

frequently update the latest news regardless of whether it is currently visible to the users, e.g., whether the display is even on or off. From the user's perspective, these downloads are not only useless, but also generate energy leaks. Eliminating these extra downloads (e.g., by only downloading when the widget is visible to users) will not influence the user's experience.

Our Automatic Detector of Energy Leaks (ADEL) system detects smartphone application energy leaks due to network downloads by tracking network data in order to determine which downloads ever have user-observable effects. We focus on network downloads for two main reasons. First, network devices, including 802.11 card and cellular devices, are power-hungry; the two devices together consume 39% of the total energy. Note that this is estimated by aggregating 137 PowerTutor users' over a months' traces [44]. Second, application developers have an especially difficult time optimizing applications for network energy efficiency due to the complicated power characteristics of the network interface [45]. As a result, a significant amount of energy is wasted due to application's network communication [45].

The definition of *energy leak* is a general definition. In practice, it would be difficult to track accurately as discussed in Section 4.5.1. However, we believe it is the perfect goal to aim at.

## 4.3 Illustrative Example and Design Challenges

We designed and implemented a system supporting on-line analysis of downloaded data objects use. We anticipate that this work will be most useful to developers attempting to improve the energy efficiency of their applications. In this section, we will first give a simple example to illustrate the use of our infrastructure. We will then discuss its design challenges.

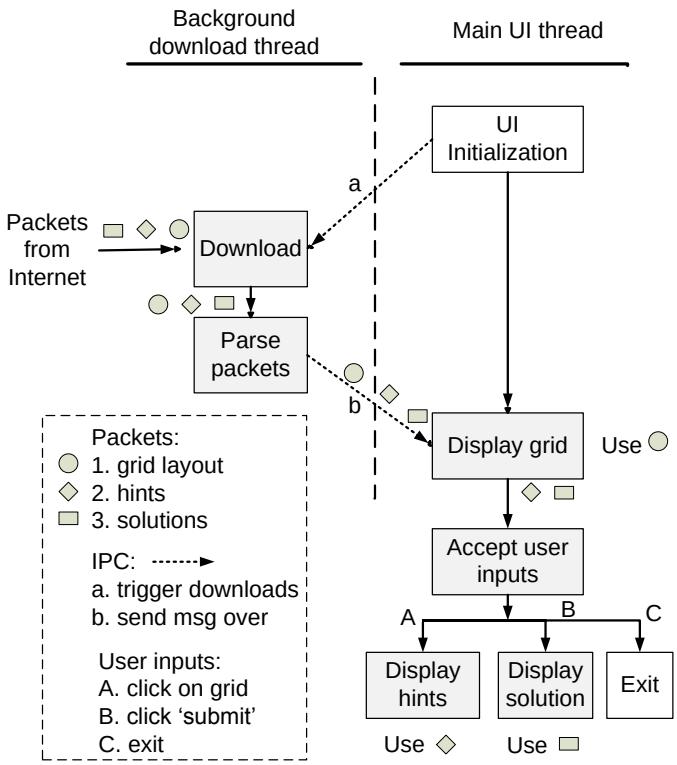



Figure 4.1: Crosswords game is with two major threads: downloading and user interface. Packets associated with tags pass through the application flow. Eventual use depends on user inputs.

### 4.3.1 Illustrative example

Imagine a game that downloads crosswords puzzles and displays them to the users. As shown in Figure 4.1, the application contains two major threads: one in charge of downloading puzzles from the Internet and one in charge of the user interface (UI) including showing the game and taking user inputs. The downloading thread receives network packets that contain the layout of grid for each game, the hints to each word, and the solution to each game. It then parses the contents of different packets and passes them as messages to the UI thread. The UI thread first displays the grid based on the message passed from the downloading thread. Then it displays hints if the user clicks on the corresponding grid. When the user finishes and submits the game, it shows the solution of the game.

To track the usage of the downloaded data objects for the crosswords game, the system

first identifies them in the network interface where a *tag* is associated with each object at the packet-level. These network interfaces are denoted as the *taint source* and the downloaded objects are *tainted* objects. As shown in Figure 4.1, each type of data objects is associated with a unique *tag*, assuming they come in separate packets. During the propagation of the *tainted* object, the system passes down the *tag* when one data object is derived from a *tainted* one. The shaded area represents the functions the *tainted* objects passing through. Finally when tainted objects reaches the *taint sink*, in this specific example, the display, they are detected and the *tags* representing the original downloaded data objects are recorded.

### 4.3.2 Design challenges

When designing a system to track the use of data objects, we were faced with the following challenges.

- *Downloaded data objects are not necessarily used immediately.* In the above example, the hints and solution of each game may be stored in memory for a long time before users reveal their inputs. This lack of temporal correlation between downloaded and eventually used rules out the possibility of some alternative approaches. For example, tracking data usage by first monitoring both the network and display interface simultaneously, then comparing downloaded content and displayed content is infeasible due to no temporal correlation. As a result, the monitoring system must be able to trace data objects through memory until they are used or deleted.

- *Data objects can be parsed or processed before they are displayed.* For example, the grid color can be received as strings in an XML file when it is first downloaded. Yet when it reaches the display, the string itself will not be displayed. Instead, it will influence the color of the output grid. Transitive derivatives of downloaded data must be considered.

- *The use of data objects depends on the dynamic environment of the application, e.g., user behavior.* As shown in Figure 4.1, the use of hints and solutions to the game depend on user behavior. As a result, static analysis of application source code is insufficient to fully understand the application's real-world data use. Therefore, we focus on dynamic analysis.

- *It is necessary to identify the original data object when a derivative is displayed.* Our goal is to help developers to (1) understand the use of each downloaded data object and (2) identify potential bugs in their code resulting in unnecessary data transmission. Therefore, it is necessary to trace downloaded data objects and their derivatives for their entire lifespans so that if they are ultimately displayed, we will be able to identify the original downloaded data objects. To achieve this, the system needs to associate a unique ID with each data object and have the ID propagated to all its derivatives. We associate unique IDs with each downloaded object and track the IDs through derivative objects.

- *Mobile devices are resource constrained.* Limits on smartphone CPU speed, memory size, and energy capacity makes high-overhead techniques such as instruction-level taint tracking [53] less practical than lower-overhead techniques.

## 4.4 System Architecture of ADEL

The challenges described in the previous section motivate us to use dynamic taint tracking analysis of downloaded data objects. The closest existing implementation we found is TaintDroid [22], an extended Android platform for improving security by identifying the flow of private information. This section first presents background information on Android that is necessary to understand ADEL, our taint tracking infrastructure. It then provides an overview of the ADEL system architecture. Finally, we explain each component of ADEL individually in detail.

### 4.4.1 Background: Android

Android is a Linux-based operating system for mobile devices such as smartphone and tablets developed by Google. It differs from a standard Linux distribution due to an additional layer of abstraction between user applications and library, the application framework layer. The four major differences between Android and standard Linux follow.

**Dalvik virtual machine, Java Native Interface (JNI), and native interface**: Android is composed of a modified Linux kernel written and compiled to native machine code, with middleware, libraries and APIs written in both Java and native code. Although development of applications using native machine code is allowed, most Android applications are written in Java.

Each Java application is compiled to bytecode and runs in the Dalvik virtual machine (DVM) and each DVM instance contains only one application. The Dalvik VM has its own instruction set. An interpreter converts Java bytecode into Dalvik code at run time. ADEL consists of taint tracking code inserted into the interpreter.

There are two types of native methods that could potentially be accessed by applications in Android: (1) native methods in the Android framework or system library such as WebKit or SQLite and (2) the native methods stored in shared object in the application package. All native methods accessed from a Java-based application must be called through JNI methods. Application packages containing native methods account for 5% of applications according to a survey of the Android market [**?**] and handling taint tracking through native code would substantially complicate the taint tracking infrastructure. Therefore, ADEL does not cover application-specific native methods in the taint tracking flow. Note, however, that it can do information flow tracking in the Java portions of applications that contain certain framework JNI methods.

**Binder**: Binder is a specially customized kernel mechanism for Android to do Inter

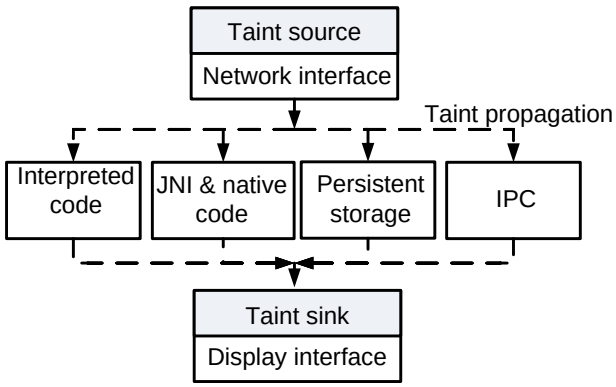



Figure 4.2: ADEL architecture.

Process Communication (IPC). In the core of binder, processes serialize the data message and pass between processes using *parcel*.

**Shared memory**: Share memory region in Android framework allows sharing between different processes. Note that reference counting in shared memory region is required in Android so that the shared region will not be removed until all processes have released it.

**Zygote process**: Zygote is the first application process loaded on top of the Dalvik VM when the system boots. It maintains memory regions that are shared by multiple applications and spawns all other Android applications by doing a regular fork().

### 4.4.2 Architecture Overview

We now provide an overview of the ADEL system architecture. ADEL provides on-line monitoring of the use of downloaded objects by applications. It uses dynamic taint tracking to follow the information flow during the entire lifespan of the downloaded object, from download until direct or transitive use. As shown in Figure 4.2, the taint flow can be decomposed into three phases: *taint source*, *taint propagation*, and *taint sink*.

**Taint source**: We identify the network interface as a *taint source*. We detect all downloaded data objects and associate a unique *tag* with each data object at packet-level. Each *tag* is a 32-bit integer that indirectly indicates (indexes) the size of packet. Note that packet size is needed to determine the number of bytes of network transmission that associated

with energy leaks. A hash table is used to map *tag*s to packet sizes. The table is kept in shared memory region and maintained by Zygote. To store *tag*s, we instrument the Dalvik VM to change the memory allocation and Java class data structures in order to store an indexing *tag* adjacent to its corresponding data object, with additional information indexed by the tag.

**Taint propagation**: During *taint propagation*, *tags* are propagated to new variables when they are derived from tainted variables. *Taint propagation* happens at four different levels: variable-level, method-level, file-level and message-level, corresponding to four different sources where tag is propagated: interpreted code, JNI and native code, persistent storage, and IPC. For interpreted code, an alternative design is instruction-level taint tracking [53]. However, this fine-grained approach induces up to 20× performance overhead for workstations and therefore is contradicted for our resource constrained mobile platform. For JNI and native code, the system retrieves the *tag*s of the inputs of the invoked methods and propagates them to the return value. This heuristic is used to prevent the overhead of monitoring native code. For persistent storage, each file is associated with a *tag*. Any write of the tainted data will taint the *tag* of the file and any data read from the file will be tainted with the same tag. Finally for IPC propagation, we associate a *tag* with the message passed between processes. Having file-level and message-level propagation indicates potential false positive in the result, yet it also covers many real scenarios. Overall, by propagating the *tag* at multiple levels, we are able to achieve system-wide taint tracking.

**Taint sink**: At *taint sink*, we identify the output of tainted data from the application. Every time a tainted object is detected, we track down and log the original downloaded object(s) that influenced the tainted object by its *tag*. Mobile computers may have multiple output device, e.g., video and audio displays. Currently ADEL only supports the video

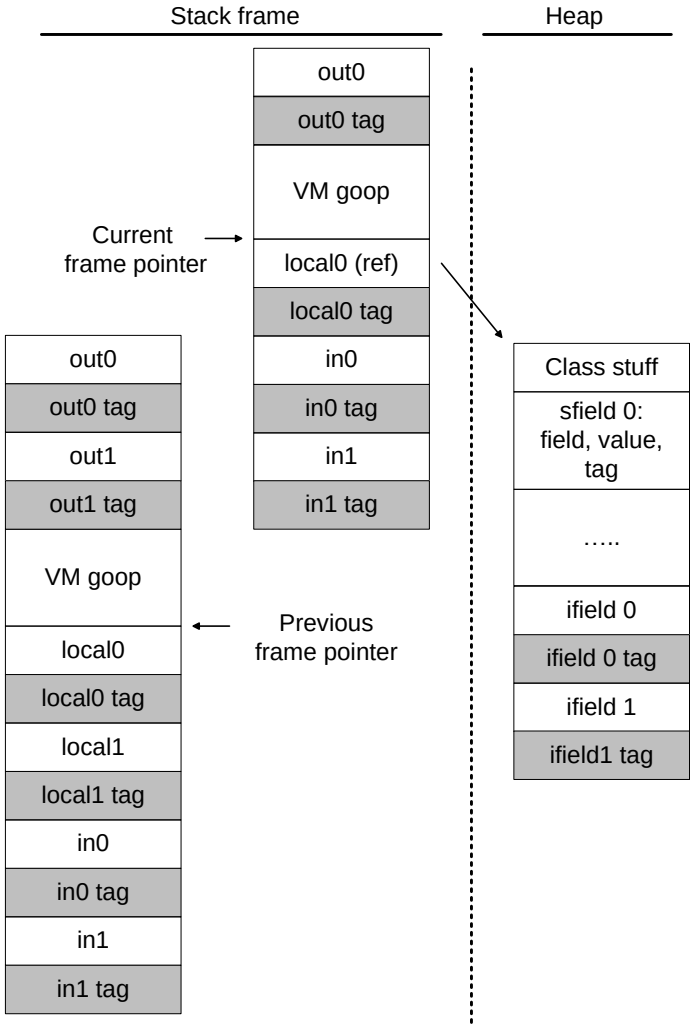



Figure 4.3: Taint tags are stored adjacent to their data objects in both stack and heap.

display interface *taint sink*, as it was used for displaying the most network-tainted data in the applications we are aware of. However, the approach could be easily extended to handle other output devices.

### 4.4.3 Implementation

We implemented ADEL on top of TaintDroid [22] with 2,414 lines of code modifications. The adaptation of this infrastructure for use in energy leak detection required changes to taint tag representation, taint propagation rules, taint source, and taint sink.

**Tag storage**

Figure 4.3 shows how ADEL stores taint tags in the Dalvik VM. Tags are associated with five types of data: local variables in method, method arguments, method return value, class static fields, and class instance fields. For local variables, method arguments and return values, whenever each new method is invoked, a new stack frame containing these data objects will be constructed. To store their tags, the size of stack frame is doubled. This allows us to store an additional 32-bit tag for each data object. As tags need to be retrieved whenever the corresponding data object is operated upon, they are stored adjacent to their corresponding data objects to allow efficient retrieval. As shown in Figure 4.3, the shaded area represents the patched memory for tags. For class static fields and instance fields stored in the heap, extra memory is used as well. The data structure of static field is modified to include a taint tag field while each instance field is extended by adding a 32-bit tag space adjacent to it. In the case of arrays and strings, only one tag is associated with the whole class object. That is to say, any value coming out of a tainted array or string is tainted. This decision was made to avoid the overhead of having multiple tags, each associating with one data element.

**Tag representation and propagation logic**

In order to identify whether there are energy leaks and what causes them, we need to track the flow of each network packet and record the contribution of tainted variables to each newly derived variable. In this case, ADEL can work backwards to determine for each packet whether it ultimately influences content displayed. One common solution in security research is to use each bit to represent one unique type of tainted variable and use OR operation on merging variables. Unfortunately, this solution only allows tracking for 32 different types of variables. In our case, each new packet represents a new tainted variable that needs to be uniquely identified. As a result, the prior solution is unable to achieve our goal.

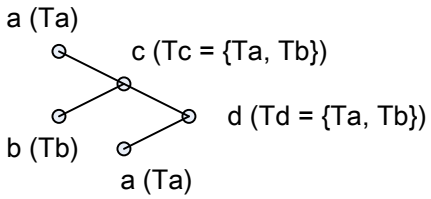



Figure 4.4: Example of correct taint tag propagation.

**Algorithm 1** Copy taint tag

**Require:** *SrcTag*

1: Increase the ref_count of *SrcTag*
2: *DestTag* $\Leftarrow$ *SrcTag*
3: **return** *DestTag*

To overcome this problem, instead of having one bit in the tag representing a unique type of tainted variable, we have each *tag* representing a set of packets that have been used to derive the corresponding variable as shown in Figure 4.4. Each packet is represented by a unique ID, *packet tag*. Every time a new variable is derived from multiple tainted variables, the union of all sets is taken to derive the new tag.

We use two hash tables in ADEL. One maps *packet tag*, the packet ID, to packet size. The other maps *tag* to the set of *packet tag* as shown below.

```
map<int packet_tag, int size>packet_map;
map<int tag, set<int packet_tag> >tag_map;
```

A new *tag* is generated if any of the following conditions holds: (1) a new data object is downloaded or (2) a new variable is derived from more than one tainted variable. The allocation of a new tag is controlled by a unique sequence number. Note that these two maps are global data structures containing all the tags for any application. Consequently, we place them in shared memory and modify Zygote to maintain the insertion of new tags and propagation of existing tags.

Taint tags are propagated using the following rules.

- Copy taint tag: When only one tainted object is used to derive a new variable, e.g.

**Algorithm 2** Merge taint tag

**Require:** *Src1Tag*, *Src2Tag*, *tag_map*

{Update an old tag or merge to create a new one}

1: **if** any of *Src1Tag* or *Src2Tag* has only one reference count **then**
2: *DestTag* ⇐ the one in *Src1Tag* and *Src2Tag* that has one tag
3: update the set of *DestTag* by inserting the set of the other source tag
4: increase the ref_count of *DestTag*
5: **else**
6: *DestTag* ⇐ a newly created tag
7: *new_set* ⇐ union of sets of both *Src1Tag* and *Src2Tag*
8: insert (*DestTag*, *new_set*) into *tag_map*
9: initialize ref_count of *DestTag*
10: **end if**
11: **return** *DestTag*

move and unary instruction, we copy the taint tag to the newly derived variable so that both variables are derived from the identical set of *packet tag*s. Meanwhile, each *tag* is associated with a reference counter to record the number of variables associated with that tag. Note that reference counting is needed for copy-on-write when merging taint tags. Algorithm 1 shows the flow of the logic.

- Merge taint tags: When more than one tainted object is used to derive a new variable, merge taint tags must take the union of the sets. This is required for any binary instruction, e.g., add. As shown in Algorithm 2, copy-on-write is used to reduce memory overhead. A new tag and set are only constructed if both of the source variable's tag have multiple references.

**Tag sources and sinks**

To determine the percentage of energy leak, ADEL needs to track the network packet from downloaded until display. Therefore, we identify the network interface as the taint source and video display interface as the taint sink. Because taint tags are stored only in the interpreted code as explained in Section 4.4.3, we choose to modify the Java system library before the JNI methods are invoked.

We instrument the receive function in the network Java library to implement network

taint sources. The granularity of tag association determines the trade off between tracking accuracy and memory overhead. Packet-level tracking is chosen over byte-level tracking to control overhead while maintaining a tolerable accuracy. Unfortunately, this granularity of tracking introduces false positives in the estimation of useful bytes and results in an underestimate in the percentage of energy leaked. Yet fortunately, despite of this, we still identify a significant amount of energy leaks in real applications as explained in Section 4.6.

Display interface is instrumented as the taint sink to detect tainted data objects. In taint sink, we identify and log tainted objects when they are rendered to the display. By tracking the tainted objects through the corresponding set of *packet tag*s, we could determine the original network packets that get displayed and determine which are useless. Android provides application developers to render the display in miscellaneous ways including various of View object and 2D drawables. These different Java library APIs eventually link to the third-party graphic libraries for rendering, e.g., Skia and WebKit. By performing a survey of the network intensive applications in Android Market, we determine that Canvas class in graphic interface and LoadListener in Webkit are the two most frequent APIs for eventual rendering use by other graphics display APIs. We therefore use these two APIs as taint sinks in ADEL. Note that WebKit is a complicated rendering engine in which data objects get passed between native methods and Java methods before they eventually get displayed. Our Dalvik-based taint tracking framework cannot trace through native methods. We instead use the following heuristic to approximate taint tracking through native graphics display routines: we assume that data objects being passed to the native WebKit rendering engine will be displayed. We have not found contradictory examples in the applications we examined; please note that perfect accuracy is not necessary for to achieve our goal of identifying and isolating energy leaks.

Table 4.1: Applications for Validation

| Application | | | Functionality |
|---|---|---|---|
| Open source | Real-world apps | Crossword | A game application that downloads crossword from multiple online sources |
| | | Read for speed | An application that downloads book and display it word by word for speed reading |
| | | Umich Busmap | A map-based application that shows the incoming buses for all bus stops |
| | | Photostream | A sample application that downloads and displays picture from Flikr |
| | Syn. app | Taint test | A self-written application that downloads and displays specific webpages |
| Closed source | ABC, Stock quote, and CNN | | Refer to the Android Market |

**Tag propagation**

Taint tags are propagated through four paths: interpreted code, JNI and native code, persistent storage and IPC.

To track through interpreted code, we modify the interpreter in Dalvik VM to realize the propagation logic. There are two alternative interpreters in the VM, one implemented in native code and the other implemented in assembly code. Dalvik VM can load either. The first works on any phone supporting native code; the second only works on the few phone models it has been customized for. The assembly-based interpreter provides higher performance. ADEL is built on the native code interpreter, i.e., it is general but there are opportunities for further improvements in performance, an issue discussed in more detail in Section 4.5.

JNI and native code taint tag propagation is handled by manually instrumenting methods on demand. This is mainly because all taint tags are stored in the interpreted code stack and therefore once a native method is invoked, the tags cannot be accessed any more. To handle taint propagation through native code, we instrument code to guarantee one post-condition: if the arguments accessed are tainted, the return value of native methods is

associated with the new tag that follows the propagation logic.

Persistent storage taint tag propagation is handled by associating each file with one taint tag. This is achieved by implementing using the extended attribute of the Android file system. That is to say, any byte read from a tainted file will be tainted with the same tag. This results in false positives, especially if the file is a database. Ideally we would like byte-level filesystem taint propagation logic. However, this would greatly increase implementation complexity and require modification of database data types [18]. This is an area for future work. Despite of the false positives resulting from this design decision, the impact on accuracy was negligible for the applications we studied: only 0.26% of network traffic ended up in database in these applications.

Message-level taint tag propagation is implemented for IPC in ADEL. This is achieved by instrumenting the *parcel* in binder as explained in Section 4.4.1. Again, false positives are induced by using message-level propagation, resulting in under estimation for energy leaks. Yet luckily even with this conservative estimate, we are still able to identify a significant amount of leaks in real applications.

## 4.5 Validation

We next examine the accuracy of ADEL and the overhead incurred by ADEL. The experiment is done on a Nexus One phone running Android 2.1 and ADEL.

### 4.5.1 Accuracy and Limitations

We examine the accuracy of ADEL by evaluating 8 applications, 5 of which are open-source and three of which are closed-source. Table 4.1 lists all the application names and corresponding functionalities. Note that we include three closed source applications for validation because we found a limited number of open-source Android applications.By analyzing each application with ADEL, we produced profiles of all network packets. This

profile contained each packet's content, its timing, and whether the packet ultimately influenced outputs to the user. Packets (transitively) reaching the taint sink were flagged as used while the rest were considered unused. We read the source code of the open-source applications, then manually examined all the network packets and used our understanding of the code to determine whether ADEL correctly labeled each. Analysis of closed-source applications required less direct confirmation. We compare the profile generated by ADEL with the content we observe on the display. For example, we manually compared the content of each network packet with the news stories displayed on the user interface for ABC news application. We identify the following types of the packet classification errors.

- **False negatives**: False negative refers to useful network transmissions that are incorrectly identified as useless. ADEL tracks data flow, not control flow. This is mainly because fully control flow tracking requires either static analysis [40] of application source code, which would undermine on-line use or incur heavy overhead [53]. One example of a false positive due to neglecting control flow follows. Many applications contain some empty HTTP responses that contain only headers and codes, e.g., "200 OK". These responses are indeed useful. They used to control the application, e.g., by using status code to select subroutine to execute branch. However, because the influence on application behavior is a result of control flow, ADEL neglects it. Luckily, the size of such packets is usually very small, resulting in only minor errors in network energy leaks. In the applications we study in Section 4.6, these empty HTTP responds account for only 0.52% of the total traffic.

- **False positives**: False positive refers to useless network transmission that is falsely identified as used. Another limitation of ADEL is the false positives induced during the tag propagation process, including packet-level tag association, array-level, message-level, and file-level tag propagation. One example we found in our experiment results from

associating a single tag with each packet. For applications in which the tagged data unit is large, e.g., a whole news article for news application, using packet-level association does not influence accuracy. However, in applications like "Umich Busmap" each logic data unit contains the text information for one bus stop and 2 to 3 data units are packed in the same packet. As a result, information on unused bus stops is falsely identified as used if any bus stop in the packet was used. These false positives could potentially be reduced by fine-grained propagation at the cost of performance overhead.

We should comment at this point that unlike the use of taint tracking in security applications, misclassification of some network traffic doesn't necessary prevent ADEL from fulfilling its design objective: to help application developers identify and isolate programming and design flaws accounting for most energy leaks. ADEL allowed the automatic identification of numerous large energy leaks in real-world energy leaks as described in Section 4.6.

### 4.5.2 Overhead Analysis

The performance overhead of ADEL results from the additional computation and storage necessary for tracking taint tags. We have two goals when designing the experiments. First, we want to understand the average slow down of the whole taint tracking flow real-world applications. Second, we want to cover all taint propagation paths to understand their overheads, allowing us to determine what sorts of slow-downs can be expected for applications with particular behaviors. To achieve these goals, we conduct two sets of experiments: one with real-world applications and one with synthetic applications targeting at stressing different propagation paths.

We use two Nexus one phones, one of which runs the stock Android 2.1 operating system, and one of which is equipped with ADEL. The portable interpreter was used on

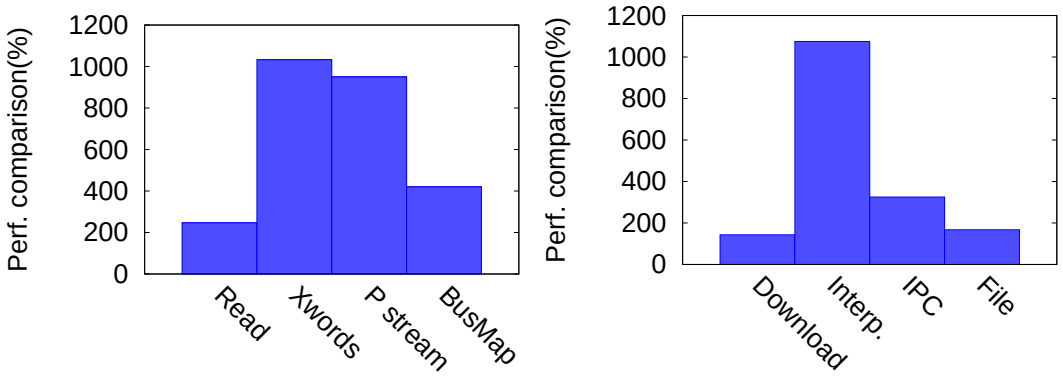



Figure 4.5: (left) Performance comparison of real apps. (right) Performance comparison on synthetic apps.

both phones (recall that ADEL is only implemented in the portable interpreter for the Dalvik VM).

**Real-world applications**: As shown in Figure IV.5(a), ADEL incurs 6.14× slow down on average for real-world applications. We notice that for both application "Cross words" and "Photo stream", where tainted objects are intensively propagated after downloaded before displayed, ADEL incurs roughly 10× overhead. For applications like "Read for speed" and "UM Bus" where tainted objects are displayed immediately after downloaded, ADEL incurs less than average slow down. Note that taint tracking generally requires significant overhead, e.g., 20 × [53].

**Synthetic applications**: To cover all possible taint propagation paths, we construct synthetic benchmarks to intensively exercise specific paths and potion in the tag flow. For example, intensive downloading is to exercise taint source. Intensive tag propagations through interpreted code, Inter Process Communication (IPC), and file are also exercised. Figure IV.5(b) demonstrates the performance overhead of ADEL for these synthetic applications. Intensive tag propagation through interpreted code incurs the most overhead, e.g., 10.75× slow down. This benchmark approaches the worst-case situation for ADEL overhead: every instruction involves tag merging. This also explains the slow down of "Cross words" and "Photo stream". There are two main sources contributing to this overhead:

context switches due to Inter Process Communication and lock acquisition due to shared memory access. Both of these results from storing tags in a shared memory region and managing them with Zygote as explained in Section 4.4. Other than the interpreted code benchmark, the slow down of all other benchmarks are relatively small.

Despite of the nonnegligible overhead, ADEL is still a useful tool as it will be used mostly by developers during application design time. One major concern about overhead is that changes in performance might change the application behavior and hence prevent us from finding energy leaks. For example, an application that uses a timer to control downloads might potentially have different behavior if latencies were to change. However, we haven't encountered such cases in our study.

## 4.6 Application Study

This section describes ADEL-assisted study of network energy efficiency for 15 Android applications. Seven of these applications are suspected to contain substantial energy leaks and we were able to manually verify the presence of significant ADEL-detected leaks in six of these. For the leaky applications, approximately 40% of network energy consumption was the result of energy leaks. We then present four root causes of these energy leaks. Our findings illustrate the value of ADEL and reveal opportunities for improving application and Android framework API energy efficiency.

### 4.6.1 Experimental Setup

We study 15 applications as summarized in Table 4.2. These applications are chosen based on two criteria: (1) there is intensive network activity, and (2) there is limited database usage in the application. We have the second criteria particularly because database usage induces false positives as mentioned in ADEL's limitation in Section 4.5. Except the five open source applications we used for validation, all closed source appli-

Table 4.2: Applications Studied

| Category | Applications |
|---|---|
| News and Weather | CNN, ABC, World News, News and Weather |
| Transportation | Google Maps, Umich BusMap |
| Game | Crossword, Read for Speed |
| Social | Facebook |
| Photography | Photostream |
| Sports | ESPN Score Center |
| Finance | Stock Quote |
| Widgets | ABC widget, Facebook widget |
| Other | Taint test |

cations are popular applications with more than 100,000 downloads in the Android Market [8].

We used ADEL to identify (and isolate) network energy leaks in each application. We are aware of two possible sources of bias in our experiments.

- *Whether an application generates energy leaks is dependent on the application's active session length.* For example, an application may download a significant amount of content right after it is launched. These downloads generate energy leaks if users exit the application immediately after launch, while they can be useful if users spend longer time in the application. To eliminate this bias, we emulate an average application session by spending more than 250 seconds in each application before exit. Note that 250 seconds represents an average application session from user study [24]. During the application session, we manually exercise the application functionalities. One exception for typical use cases is widget applications. Widgets typically stay active without exiting. We therefore attempt to approximate typical use of two widget applications by placing them on the home screen, reading updates from them every five to ten minutes, and monitoring their activities for 30 minutes for each application.

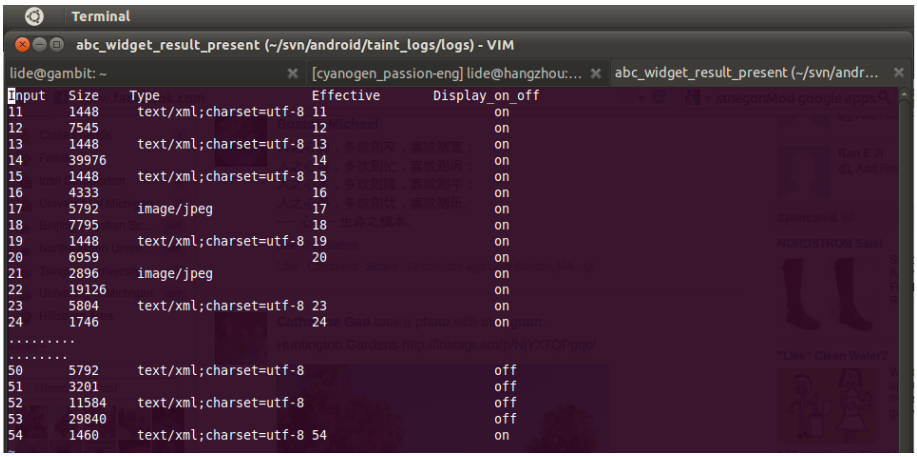
- *The propensity of an application to leak energy might depend on whether it is*

*in its initialization or normal use phase.* Imagine a news application that updates latest articles during the first use of the application in a day and store them for later use. During a single session, a user might not consume all the articles, making the unused articles appear useless. However, the user might ultimately read these articles in a later session. As a result, analysis of the first session might result in overestimation of energy leaks. To solve this problem, we run each application three times, with a five-hour intervals between runs.

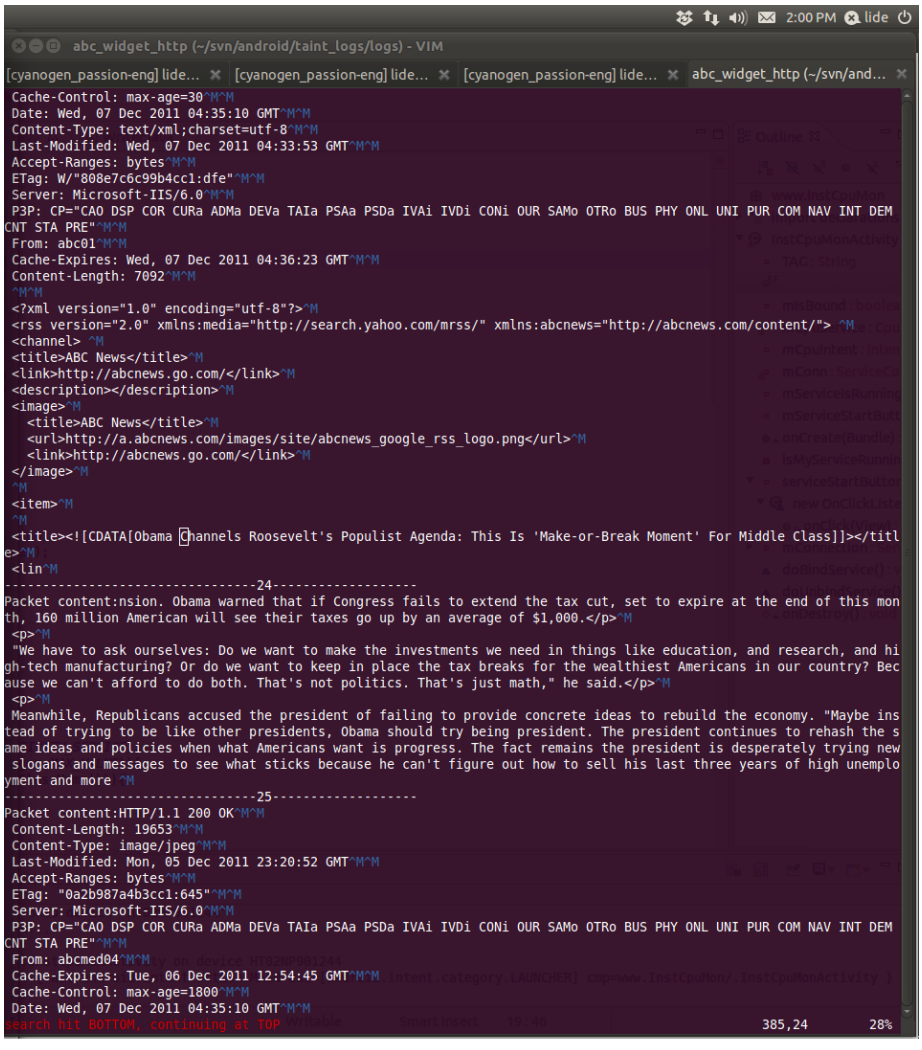
To identify the root cause of energy leak, we leverage on the information ADEL provides together with manual inspection of the source code of the application if provided. ADEL provides two major types of information to developers as follows. (1) Packet usage detected by ADEL. For example, as shown in Figure IV.6(a), ADEL labels each packet with a unique ID and associate the ID with packet size, type, whether it is shown on display (effective), and whether the display is on or off when it is being shown. This information helps developer to know the percentage of useless packet and also understand the sources of energy leaks. (2) The packet content associated with its ID. This information enables developer to understand what the useless packet contains and potentially indicates the solution to eliminate such packets.

### 4.6.2 Findings

Figure 4.7 shows the percentage of *useful network transmission* of each application, where *useful network transmission* refers to network packets that ultimately influence the content displayed. We derive this ratio for each application by averaging traces from three runs. A higher percent of useful transmission suggests a more efficient design. As shown in Figure 4.7, the applications with lower bars on the left are suspected applications with higher energy leaks. Interestingly, there is a clear cut between the suspected leaky appli-




(a) Result showing the usage for each network packet.




(b) Content contained in each network packet.

Figure 4.6: Information ADEL provided to developers.

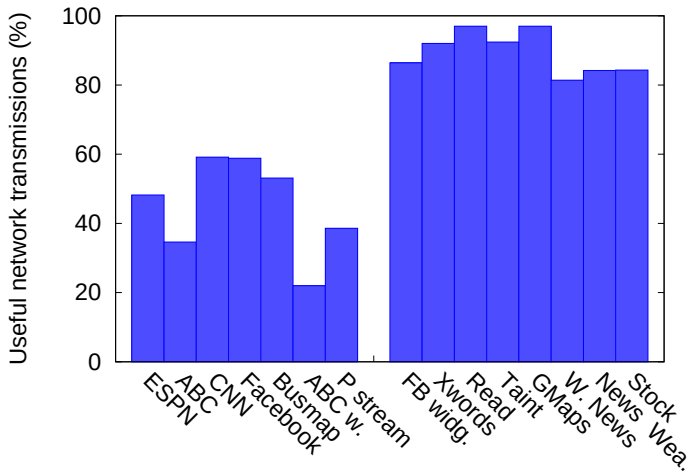



Figure 4.7: Percent of useful network transmissions for applications separated by efficiencies, for leaky (left) and efficient (right) applications.

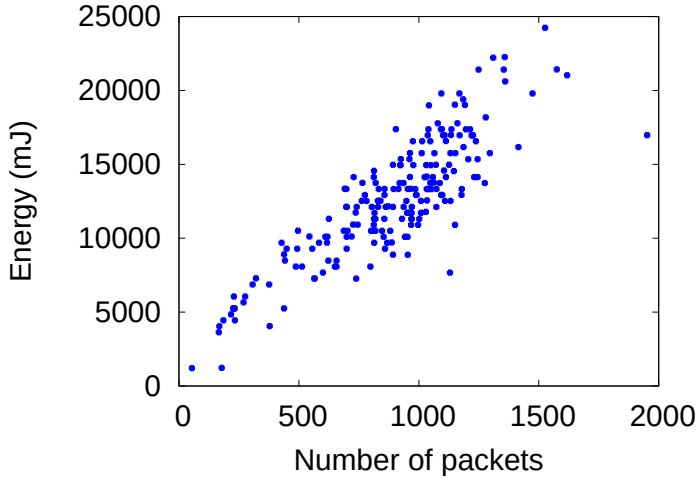



Figure 4.8: Correlation between network energy and network transmission.

cations and efficient applications at 60%. To estimate the energy reduction of eliminating these energy leaks, we examine the correlation between energy and total number of packets transmitted. As shown in Figure 4.8, energy consumed by network is roughly linearly correlated with total number of packets sent with a correlation coefficient of 0.86. Note that this figure is obtained by monitoring one user's total network usage and energy consumed using PowerTutor over a month. With this correlation, overall, reducing the energy leaks detected by ADEL could save 56.5% of energy due to network communications for the detected leaky applications.

In order to understand the root causes of the suspected energy leaks, we examine the

Table 4.3: Root Causes for Leaks

| Causes | | Applications |
|---|---|---|
| Design flaws | Misinterp. API | Photo Stream |
| | Download scheme | Umich Bus, and ABC widget |
| | Repetitive download | ABC widget |
| Aggressive prefetching | | ESPN, ABC, CNN, and Facebook |

network packet usage profile generated by ADEL for each leaky application. The profile contains each network packet's taint tag, packet content, packet arrival time, and whether the packet leads to content displayed in the end. We also manually go through their source code if available to confirm our suspicions. We identify four major reasons for energy leaks: (1) misinterpretation of callback API semantics, (2) poorly designed downloading scheme, (3) repetitive downloads, and (4) overly aggressive prefetching. We associate applications with their causes of energy leaks in Table 4.3. Note that a single application may have multiple types of energy leaks. We categorize the first three causes as design flaws and the last as (overly) aggressive prefetching.

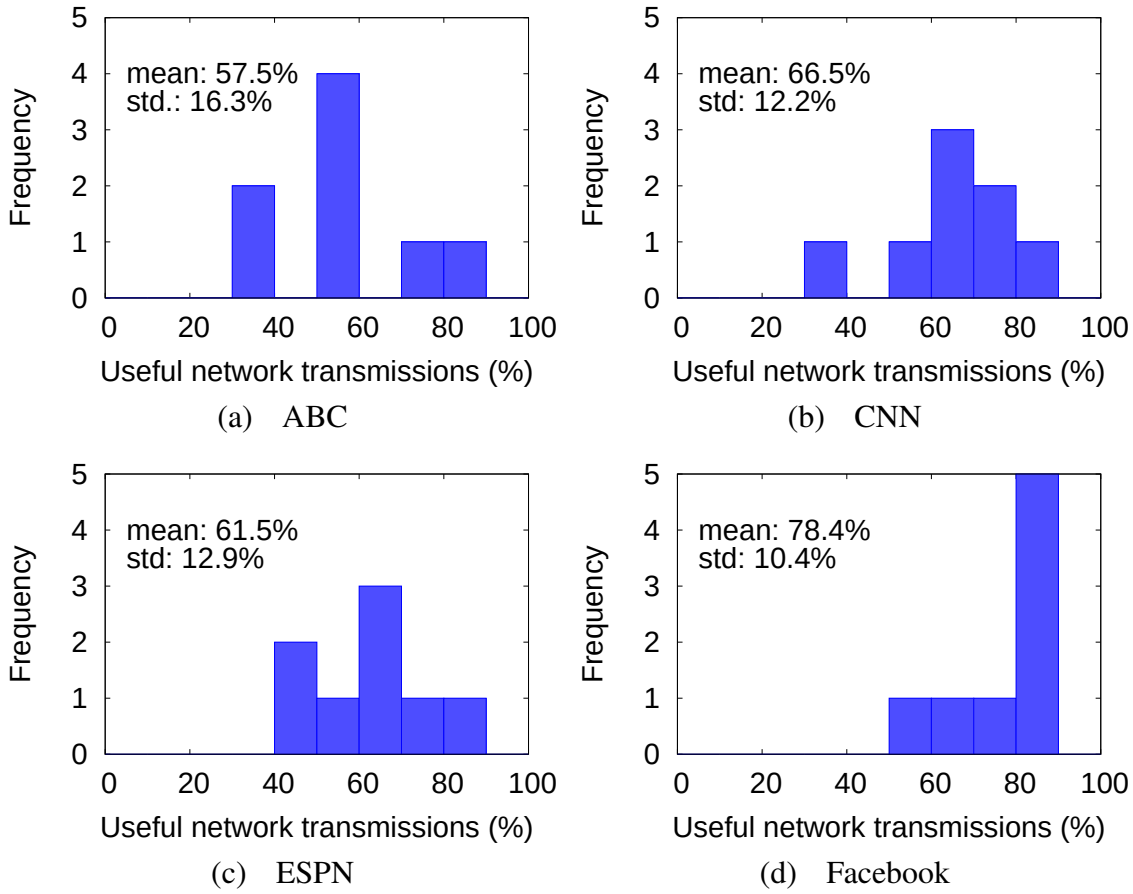
### 4.6.3 Application Design Flaws

This subsection, expands on the three types of unambiguous design or programming errors that resulted in network energy leaks. Section 4.6.4 will deal with energy leaks caused by (overly) aggressive predictive prefetching.

**Misinterpretation of callback API semantics**:

We find that the application "Photostream" incorrectly uses APIs, leading to substantial network energy leaks. "Photostream" is a sample application written by Google that intends to show beginner developers how to use specific APIs. The application allows users to retrieve other user's pictures from Flickr. As shown in Figure 4.7, only 38.6% of network transmissions are used to influence the display.

Investigating the content and timing of unused network packets reveals that the "Photo-




(a) ABC (b) CNN

(c) ESPN (d) Facebook

Figure 4.9: Histogram of percent of useful network transmission from user study for suspected leaky applications.

Stream" application downloads many unused packets even when the application is put into the background. By exploring the source code, we confirm that the background downloading thread is stopped in *onDestroy()*, a callback method that is not guarantee to be called even after the application is put into background. As a result, the application continues downloading objects that cannot ever be displayed, resulting in network energy leaks.

This bug can influence many applications due to misunderstanding of the documentation. The Android SDK documentation clearly suggests that developers to clean up threads in the *onDestroy()* callback method right before the application is about to be killed. This suggestion is reasonable for handling threads in general because it reduces the cost of initializing a new thread when the application is resumed. However, in this particular case, it

produces energy leaks because the thread continues downloading (useless) data. To make things worse, the source code of "PhotoStream" was provided as an example application for use by beginner Android developers, propagating the design error into other applications. Note that this bug could be easily fixed by stopping the thread in *onPause()*, which is guaranteed to be called right after the application is placed in the background.

**Poorly designed downloading scheme**: Poorly designed downloading scheme refers to frequent updates without considering either device state (e.g., whether the screen is on or off) or user input. As shown in Table 4.3, we identify two applications with this problem: "ABC widget" and "Umich BusMap". The first wastes 78.0% of network transmissions and the second wastes 46.9% of network transmissions.

"ABC widget" is the news widget application that can be placed on home screen. Surprisingly, by examining the timing of unused network packets, we find that the widget keeps updating news article every 5 minutes, regardless of whether the widget is being shown or even whether the display is on or off. This poorly designed downloading scheme is detrimental to smartphone battery life as it not only wastes energy consumption on network and CPU when the phone is active, but also triggers the phone to wake up every 5 minutes even when it is idle.

More interestingly, further exploration on the widget related APIs in Android leads us to believe that the Android framework fails to provide a good mechanism for widgets to stop updating when not visible. There are three scenarios when a widget is not visible: the display is off, the widget is placed on another home screen that is not displayed, or another application is running in the foreground. Android's API provides an indirect way to help the widget to identify the first scenario [1], while it is impossible for widget to identify the latter two. Apparently, "ABC widget" even fails to identify the first scenario. This finding suggests necessary improvements in the Android framework API, e.g., providing widget

applications notifications when they are not visible or stop sending the update notifications when widgets are not visible.

"Umich BusMap" contains another example of a poorly designed downloading scheme. It is a map-based application that informs users to the next three school buses to arrive at bus stops. A user selects specific stop to view its bus schedule. The schedule for only a single stop can be examined at any particular time. Examination of the timing and content of unused network packets reveals a cause for energy leaks: regardless of which stop the user selects, the application downloads schedules for all stops, even those that are not visible on the current map. To eliminate the inefficiency, developers should design downloading schemes that take possible (and ideally common) user input into consideration.

**Repetitive downloads** "ABC widget" application is found to have repetitive downloads, e.g. identical copies of same packets are downloaded and yet never used. By examining the content of its unused network packets, we discover that every time the application updates, it downloads all recent news articles, even when they have not been updated. Clearly, this observation suggests that the application naïvely fetches the latest news articles from Internet without caching any of the downloads. In order to fix this bug, application developers could add an expiration time for each article or have the application server notify the client of new updates.

### 4.6.4 Overly Aggressive Prefetching

Prefetching has been used extensively in both desktop and mobile environments to improve user satisfaction by eliminating download delays for data that are expected to be required soon. It generally uses prediction, often of future user actions. Such prediction cannot be perfectly accurate. Therefore, even a well-developed predictive prefetching application will download some objects that are never used. In order to enable a rational

understanding of the trade off between response time and unused transmission, developers must have the unused transmission of the target application to start with. ADEL helps to provide this number and hence also assists developers to determine when it would improve application energy efficiency by (possibly on-line) adjustment of prefetch aggressiveness and/or prefetching predictor accuracy. As shown in Table 4.3, four out of seven suspected leaky applications we studied generate energy leaks due to significant wasted prefetching.

Whether an application generates significant wasted prefetching cannot be determined by a single user's trace. This is mainly because an unnecessary packet for one user may be useful for another user due to varying use patterns. Hence, to capture the prefetching aggressiveness of an application, it is necessary for us to study the applications with normal usage patterns.

We conducted a small-scale user study with 8 graduate students who are regular smartphone users. Note that none of the users were told the intention of the experiment in order to prevent their behavior from being biased. Each participant was asked to use the four leaky applications as they would use in daily life. There was no time requirement for the use of any particular application. After gathering user traces for one application, the percent of *useful network transmission* is derived from each trace. Note that this ratio directly indicates prefetching aggressiveness because they are inversely correlated: a smaller percent of useful network transmission suggests more aggressive prefetching schemes. Figure 4.9 summarizes the distribution of the percent of *useful network transmission* for each application.

We can draw two observations from Figure 4.9.

- The percent of *useful network transmission* from each user varies significantly for one application due to user-dependent variation in application use. As shown in Figure 4.9, the percent of *useful network transmission* spans a range of more than 40%. This substan-

tial variation among users suggests that the optimal prefetching aggressiveness depends on the user; it suggests that a static prefetching scheme is unlikely to be optimal for all users.

- "Facebook" is more efficient than the other three suspected leaky applications. For example, more than 80% of the network transmissions are useful for 5 of the 8 users. What's more, the mean of the distribution is 78.4%, which indicates that 78.4% the downloaded content turns out to be useful for an average user. As a result of this, we find "Facebook" not leaky, despite of our previous suspicion. This difference in suspected candidates and confirmed leaky applications further reassures the necessity for the user study.

Overall, based on our observations, the applications "ABC", "CNN", and "ESPN" should reduce their prefetching aggressiveness to enable a more energy efficient design.

ADEL's ability to determine prefetching aggressiveness enables three additional use scenarios in addition to detecting energy bugs. First, developers can use ADEL to learn the effectiveness of the prefetching scheme of their target applications and adjust during application design. Second, users can occasionally use ADEL to determine whether an application is prefetching with appropriate aggressiveness and adjust this parameter within the application or operating system. Third, with limited performance optimizations of ADEL, it could potentially be used within the regular operating system to provide applications with a real-time feedback on prefetching energy waste for one particular user. Applications could then adjust their prefetching aggressiveness to customize the user's current behavior.

## 4.7 Summary

In this work, we provided a definition of *energy leaks*: common but hard to automatically detect energy waste resulting from useless activities in smartphone applications. We

have described the design and implementation of ADEL, an Automatic Detector for Energy Leaks. ADEL identifies unnecessary network communication and its root causes via dynamic taint tracking. ADEL incurs 21.8% performance overhead on average.

We studied 15 real-world applications using ADEL and found that 6 of these have significant energy leaks. Eliminating these energy leaks results in an average 56.5% reduction in energy consumption due to network communication. Our study revealed four common causes for energy leaks: misinterpretation of callback API semantics, poorly designed downloading scheme, repetitive downloads, and (overly) aggressive prefetching. Both ADEL and our study of network energy leaks in Android applications have the potential to help application developers improve energy efficiency.

# CHAPTER V

# Related work

## 5.1 Power Model Construction

This section describes related work on both online power modeling and smartphone use analysis. We will summarize these works separately.

**Power modeling:**Power modeling has been well studied by many researchers, both for mobile embedded systems and general-purpose computers. Some power modeling techniques [32, 33] require deep knowledge of the relationship between processor functional unit activities and their resulting power consumptions. Run-time functional unit activities are monitored using built-in hardware performance counters. In contrast, other researchers developed "black-box" microprocessor power models [13, 17] that require no knowledge of hardware component implementation. These models are based on the assumption of linear relationships between processor power consumption and several hardware performance counters, e.g., instructions executed and translation lookaside buffer misses. Flinn and Satyanarayanan [25] developed a workstation power modeling technique that assigns energy consumption to processes or procedures within a process. Such models are simple, fast, and impose low overhead. However, they only model power consumed by the CPU and therefore provide only part of the solution for embedded system power estimation.

Mobile embedded system power models are generally component-based. Cignetti et

al. proposed a full-system power model for Palm personal digital assistant [16] and Shye et al. [50] derived a system-level power model for Android platform smartphones. Both power models were constructed by correlating operating system visible state variables with power consumption while running a range of normal software applications. This modeling technique is sometimes accurate. However, it suffers from a potential drawback: the accuracy of the resulting model relies on the training applications exercising the full set of component activity and power management states that may be encountered during the use of model. We suggest, instead, that training and characterization applications be designed to explicitly exercise all relevant system states, so that the resulting model is appropriate for use with arbitrary applications. Section 2.2.2 describes the selection of components and states to consider.

Application-based power modeling has also been studied on mobile systems. Negri et al. [41] proposed a Finite-State-Machine (FSM) based approach that models each application as a FSM and the power consumption of each state is measured. However, this work requires off-line characterization of each application to understand the FSM for modeling and therefore cannot handle a large amount of applications. Our work differs by requiring no additional understanding of the application's property.

The above power models are constructed using external power meters. To the best of our knowledge, only two papers have proposed battery behavior based power model construction techniques. The concurrent work from Dong and Zhong [20] proposed an automatic construction of power model using a smart battery interface, while Gurun and Krintz [27] proposed an adaptive power estimation model that uses the built-in Battery Monitor Unit (BMU). Both techniques require knowledge of the discharge current and remaining battery capacity, which are not available for most phones. Our technique relies only on knowledge of the battery discharge voltage curve and access to a battery voltage

sensor, which is available on most smartphones.

## 5.2 Characterization of Real-world Workload

This section summarizes characterization of real-world workload.

A number of commercial tools have been released for bug detection. The default mobile platform bug report system including iOS, Android, and Windows, only report application crash logs to developers. Yet it is often not that useful because the log rarely contains the condition that triggers the crash, e.g., different network environment. Moreover, even when the bug reports are helpful to fix crashes, it rarely reports any performance inefficiencies that the user experiences. Another commercial tool that is also available to monitor mobile applications' usage behavior in the wild is called Flurry [4]. This tool reports application launches, users' session length and geographic information of users. None of these tools provide information that is fine-grained enough for developers to track user transactions and detect performance inefficiencies.

There are also research works that have collected user traces from the wild to detect inefficiencies in energy, network or application behavior. Carat [3] is a tool that collects energy usage in the wild while 3GTest is a tool that focuses on network usage. Falaki et al. [24] discovered the diversity in user behaviors and suggested user-adpative mechanisms to improve energy efficiency and user experience. Banerjee et al. [10] summarized users' battery charging behavior.

## 5.3 Performance Monitoring and Optimization

Panappticon detects and monitors perceived user transactions of mobile application experienced in the wild. It further detects system and application inefficiencies and make suggestions. This section summarizes prior works on the following related topics: perfor-

mance monitoring and debugging of user transactions, monitoring mobile applications in the wild, and characterization of mobile workloads.

**Performance monitoring and debugging for user transactions or request:** A number of works have been proposed for performance monitoring and debugging for user transaction or request, especially for distributed systems. These prior works can be placed into two major categories, as summarized as follows.

The first category of works require developers' detailed knowledge of applications semantics or source code to capture user transactions [11, 34]. Magpie [11] is a system that monitors and characterizes user requests for servers. It leverages the event semantics provided from the developers to join the events logged in the system together into transactions. Magpie is instrumented on Windows platforms. Different from Magpie, Panappticon does not require developers' input, which eases the burden for developers. Also note that Magpie focus on server workloads while Panappticon focus on mobile platform. This difference in platforms leads to different system design.

Another work in the first category is LagHunter [34]. It is a debugging tool that identifies perceptible performance bugs by monitoring application behaviors. To identify user transactions, it requires developers to provide a set of landmark methods, which are normally the methods that handle UI inputs. By inserting LagHunter code in such methods, the call stack of each landmark method can be tracked. Unlike Panappticon, the approach in LagHunter only allows tracking synchronized UI event handling. This limitation significantly constrains its usage on platform that allows multithreading like Android, where most complicated transactions are handled asynchronously.

In contrast, the second category of works treat the applications as black-box and do not require developer's input [48]. AppInsight is a system that instruments mobile application binaries to automatically identify the critical path in user transactions. Similar with our

approach, AppInsight is also using causality between different work unit across threads to identify user transactions. However, different from our work, AppInsight does not record any kernel event or framework activity and therefore won't be able to catch any resource dependency between threads and detect system-wide inefficiencies.

**Characterization of mobile workloads performance:** Another goal of Panappticon is to understand the impact of architectural changes on perceived user transactions so that we could identify trends to guide future design. To this end, the closet work done by Gutierrez and et. al. [28] characterized the microarchitectural behavior of a set of representative smartphone applications and constructed a mobile benchmark based on these characteristics. They found that instruction cache miss rate of the smartphone applications is higher compared with SPEC and therefore suggest a bigger instruction cache. They also observed potential opportunities to improve CPU performance by integrating more capable branch predictors. Our work differ with their work by having different purpose and focusing on user transaction only instead of performance in general.

## 5.4 Energy Debugging and Optimization

Our work in Chapter IV uses dynamic taint analysis to identify energy leaks due to network communication. This section summarizes prior work on the following related topics: energy debugging, network energy reduction, and taint analysis.

**Energy debugging:** Energy debugging is an emerging research area, with a limited number of publications so far. The work by Pathak et al. [43] is closest to ours. They provide a taxonomy of energy bugs by examining posts from mobile user forums and operating system bug repositories, but do not provide solutions for finding and correcting energy bugs. They categorize energy bugs into four classes: hardware-related, software-related, external-condition-triggered, and those of unknown causes. Our work provides a

solution for detecting and isolating energy leaks, which covers a subset of energy bugs in their taxonomy.

**Reducing network power consumption:** A lot of work focuses on reducing energy consumption due to network communication. This work can be placed in two major categories: (1) adapting power management scheme to network traffic and (2) adapting the traffic to power management scheme. Most prior work assumes that transmitted data are useful. We question this assumption and detect network energy waste due to transmission of useless data. Existing network energy consumption optimization techniques can be applied together with our work: they are orthogonal and in some cases synergistic.

Techniques have been proposed to adapt network power management to application traffic [7, 36, 46]. Krashinsky and Balakrishnan [36] propose to adapt network interface sleep durations depending on past application network activity. This allows the network interface to sleep for longer periods of time when there is no activity, thereby reducing the energy consumed receiving beacons sent from the access point. Anand et al. [7] describe a self-tuning power management scheme that provides a simple interface allowing applications to disclose hints about their intent in using the network interface. The power management strategy adapts to observed network access patterns.

Another class of network energy reduction techniques adapt network traffic to the power management scheme. These approaches use an additional support device to shape the network traffic. Some used an additional proxy server [9, 31, 49]. Armstrong et al. [9] proposed to shift the polling responsibility from the mobile client to a proxy server so that network workload is sent and received in batch. This approach prevents the network card from frequent transition between power states and hence allows the network card to sleep longer between transitions. Xie et al. [51] had the wireless access point select certain parameters for its clients, such as beacon interval, listen interval and contention window

size, in order to reduce the simultaneous wake-up events of clients.

**Taint analysis:** Our energy leak detections and isolation framework uses dynamic taint analysis. In particular, we build upon TaintDroid [22], an extended Android platform that supports system-wide taint tracking through the Dalvik Virtual Machine (DVM) and persistent storage, e.g., in files. TaintDroid detects security flaws in applications for mobile devices by associating tags with sensitive information, e.g., location information or phone contacts. Information leaks are detected when sensitive information leaves the mobile devices, e.g., via the network interface. Our work differs by solving the very different problem of identifying communication energy leaks. This new problem definition requires changes in the representation of tags and method of tracking information flow (from network interfaces to eventual display or deletion). To the best of our knowledge, this is the first time taint tracking analysis is used to identify energy leaks.

There are other methods of dynamic taint analysis in virtual machine and interpreter environments [15, 29, 42, 47, 52]. Haldar et al. [29] instrumented the Java String class with taint tracking and Xu et al. [52] automatically instrument the PHP interpreter source code for dynamic information tracking. These papers all solve entirely different security-related problems from our work, but their implementations are related because all use taint tracking.

# CHAPTER VI

# Conclusion and Future Work

My thesis is dedicated to provide a practical, automatic, efficient, and effective framework to help mobile system and application developers to understand, monitor, and optimize the energy consumption and performance of their designs. My contributions can be summarized as follows.

- *Energy related:* We proposed an automatic power model construction technique to derive the power model of hardware components. This is the first time an automatic model construction technique is proposed. We also developed PowerTutor, the first tool that monitors the real-time energy consumption of the different hard components and different applications. Equipped the PowerTutor, we performed the first study on energy usage by analyzing hundreds of unique users' over one year's traces. As far as we know, this is the first study on energy usage with this scale. One key observation revealed by the study is that display and network interfaces are the two top most energy hungry hardware components.

- *Performance related:* We identified *perceived user transaction* as the performance metric for mobile system and applications. We provided Panappticon, a light-weight, system-wide, fine-grained event tracing system that automatically identifies critical path in such transactions. This is the first Android-based framework that monitors perceived user

latency. We also deployed Panappticon on real users. Our traces collected from Panappticon reveals performance inefficiencies in applications, operating system policy.

- Having the knowledge from both energy and performance, we proposed energy optimization technique that maintains user perceived experience. We observed that operating system and applications frequently consume energy to perform tasks that are ultimately useless, a phenomenon we call *Energy Leaks*. We then proposed and implemented ADEL (Automatic Detector of Energy Leaks), the first diagnosis framework that detects and isolates energy leaks resulting from unnecessary network communication. ADEL revealed that many popular applications built by professional developers have significant network usage inefficiencies that previously are unknown.

In summary, my disseration indicates the importance of the following rules for mobile system and application developers:

1. Understanding the energy and performance indication for design decisions;
2. Identifying optimization opportunities after measuring real-world behavior;
3. Balancing the tradeoff between energy and performance when developing optimization strategy.

## 6.1 Future work

There are a number of natual future directions I'm interested in exploring:

- **Fine-grained power modeling:** Providing developers code-level power consumption and giving concrete suggestions for energy optimization is the ultimate goal for power modeling. To achieve this, there are two major challenges: (1) how to get fine-grained power measurement? On nowadays processor, each method in the source code can be executed more than a million times per second. This presents a huge challenge for power modeling. (2) How to attribute this fine-grained measurement to the actual code? Many

execution is done via asynchronous fashion and this breaks the temporal correlation between the power measurement and the code being executed. I'm interested in pursuing this direction to overcome these two challenges.

- **The optimization techniques:** Besides network, I plan to pursue other hardware components to reduce the total energy consumption while maintaining user experience, in particular, CPU and the OLED display. These three components together are the top three energy hungry devices in smartphone today. For CPU, we face the similar challenge for network: how to track unnecessary computation? This can be solved by using similar taint-tracking infrastructure. OLED presents a different challenge: how to convert energy-inefficient colors into energy efficient ones without influencing user's perceived experience?

# APPENDIX

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] Android SDK reference. http://developer.android.com/reference/packages.html.

[2] Application breakdown by category. http://www.gottabemobile.com/2011/07/06/ipad-app-store-breakdown-top-apps-categories-chart/.

[3] Carat. http://carat.cs.berkeley.edu.

[4] Flurry. http://www.flurry.com/.

[5] Galaxy Nexus. http://www.google.com/nexus/.

[6] US smartphone users now over 100 million. http://www.digitaltrends.com/mobile/us-smartphone-users-now-over-100-million-android-increases-market-share/.

[7] M Anand, E Nightingale, and J Flinn. Self-tuning wireless network power management. In *Proc. Int. Conf. Mobile Computing and Networking*, pages 176–189, September 2003.

[8] Android market. http://market.android.com.

[9] Trevor Armstrong, Olivier Trescases, Cristiana Amza, and Eyal de Lara. Efficient and transparent dynamic content updates for mobile clients. In *Proc. Int. Conf. Mobile Systems, Applications And Services*, pages 56–68, June 2006.

[10] N Banerjee, A Rahmati, M D Corner, S Rollins, and L Zhong. Interactions and adaptive energy management in mobile systems. September 2007.

[11] P. Barham, A. Donnelly, R. Isaacs, and R.Mortier. Using magpie for request extraction and workload modeling. In *Proc. Int. Symp. Operating Systems Design and Implementation*, June 2004.

[12] Battery. Battery and energy characteristics, 2005. http://www.mpoweruk.com/performance.htm.

[13] F Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proc. Special Interest Group on Operating Systems European Wkshp.*, pages 37–42, 2006.

[14] I. Ceaparu, J. Lazar, K. Bessiere, J. Robinson, and B. Shneiderman. Determining causes and severity of end-user frustration. In *International Journal of Human-Computer Interaction*, 2004.

[15] D Chandra and M Franz. Fine-grained information flow analysis and enforcement in a Java Virtual Machine. In *Proc. Annual Computer Security Applications Conference*, pages 463–475, December 2007.

[16] T. Cignetti, K. Komarov, and C. Ellis. Energy estimation tools for the Palm. In *Proc. Int. Wkshp. on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 96–103, August 2000.

[17] Gilberto Contreras, Margaret Martonosi, Jinzhan Peng, Roy Ju, and Guei-Yuan Lueh. XTREM: a power simulator for the Intel XScale. In *Proc. Conf. Languages, Compilers, and Tools for Embedded Systems*, pages 115–125, June 2004.

[18] B. Davis and H. Chen. DBTaint: cross-application information flow tracking via databases. In *Proc. USENIX Conf.*, pages 12–12, June 2010.

[19] M Dong and L Zhong. Power modeling for graphical user interfaces on OLED-based displays. In *Proc. Design Automation Conf.*, July 2009.

[20] M Dong and L Zhong. Sesame: A self-constructive virtual power meter for battery-powered mobile systems. Technical report, 2010.

[21] W Enck, P Gilbert, B Chun, L Cox, J Jung, P McDaniel, and A Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. Int. Symp. Operating Systems Design and Implementation*, October 2010.

[22] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, and J. Jung. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. USENIX Conf.*, June 2010.

[23] H. Falaki, D.Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proc. SIGCOMM Conf. on Internet Measurement*, 2010.

[24] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proc. Int. Conf. Mobile Systems, Applications And Services*, pages 179–194, June 2010.

[25] J Flinn and M Satyanarayanan. PowerScope: a tool for profiling the energy usage of mobile applications. In *Proc. Wkshp. on Mobile Computer Systems and Applications*, page 2, 1999.

[26] E. Group. Environment working group data, 2010. http://www.ewg.org/cellphoneradiation/Get-a-Safer-Phone?&allavailable=1&order=sar.

[27] S Gurun and C Krintz. A run-time, feedback-based energy estimation model for embedded devices. In *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, pages 28–33, October 2006.

[28] Anthony Gutierrez, Ronald G. Dreslinski, Thomas F. Wenisch, Trever Mudge, Ali Saidi, Chris Emmons, and Nigel Paver. Full-system analysis and characterization of interactive smartphone applications. November 2011.

[29] V Haldar, D Chandra, and M Franz. Dynamic taint propagation for Java. In *Proc. Annual Computer Security Applications Conference*, pages 303–311, 2005.

[30] Harri Holma and Antti Toskala. *HSDPA/HSUPA for UMTS: High Speed Radio Access for Mobile Communications*. John Wiley & Sons, 2006.

[31] B.C. Housel and D.B. Lindquist. WebExpress: a system for optimizing web browsing in a wireless environment. In *Proc. Int. Conf. Mobile Computing and Networking*, pages 108–116, June 1996.

[32] Canturk Isci and Margaet Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proc. Int. Symp. Microarchitecture*, pages 93–104, December 2003.

[33] Russ Joseph and Margaret Martonosi. Run-time power estimation in high-performance microprocessors. In *Proc. Int. Symp. Low Power Electronics & Design*, pages 135–140, August 2001.

[34] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *Conf. on Object-Oriented Programming System, Language, and Applications*, October 2011.

[35] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *USENIX Conference on File and Storage Technologies*, February 2012.

[36] R Krashinsky and H Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *Proc. Int. Conf. Mobile Computing and Networking*, pages 135–148, September 2002.

[37] D Linden and T. B. Reddy. *Handbook of Batteries*. MacGraw-Hill, 2002.

[38] Monsoon. Monsoon power monitor, 2008. http://www.msoon.com/LabEquipment/PowerMonitor/.

[39] MSM700. MSM7000 chipset, 2010. http://www.qualcomm.com/products_services/chipsets/index.html.

[40] A C Myers. JFlow: practical mostly-static information flow control. In *Proc. of the ACM Symp. on principles of Programming Languages*, pages 228–241, January 1999.

[41] L Negri, B Domenico, and F William. Application-level power management in pervasive computing systems: a case study. 2004.

[42] J Newsome and D Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. February 2005.

[43] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proc. Wkshp. Hot Topics in Networks*, November 2011.

[44] PowerTutor. PowerTutor, 2009. http://powertutor.org.

[45] F Qian, Z Wang, A Gerber, Z M Mao, S Sen, and O Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proc. Int. Conf. on Mobile Systems, Applications, and Services*, June 2011.

[46] D. Qiao and K. Shin. Smart power-saving mode for IEEE 802.11 wirelss LANs. In *Proc. Int. Conf. Computer Communications*, pages 1573–1583, March 2005.

[47] F Qin, C Wang, Z Li, H S Kim, Y Zhou, and Y Wu. LIFT: a low-overhead practical information flow tracking system for detecing security attacks. In *Proc. Int. Symp. Microarchitecture*, December 2006.

[48] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: mobile app performance monitoring in the wild. In *Proc. Int. Symp. Operating Systems Design and Implementation*, October 2012.

[49] M C Rosu, C M Olsen, C Narayanaswami, and L Luo. PAWP: A power aware web proxy for wireless lan clients. In *Workshop on Mobile Computing Systems and Applications*, pages 206–215, December 2004.

[50] A Shye, B Scholbrock, and G Memik. Into the wild: studing real user activity patterns to guide power optimizations for mobile architectures. In *Proc. Int. Symp. on Microarchitecture*, pages 168–178, 2009.

[51] Y Xie, X Luo, and R Chang. Centralized PSM: an AP-centric power saving mode for 802.11 infrastructure networks. *Proc. Int. Conf. on Sarnoff symposium*, pages 375–379, 2009.

[52] W Xu, S Bhatkar, and R Sekar. Taint-enhanced policy enformacement: a practical apporach to defeat a wide range of attacks. In *Proc. USENIX Conf.*, pages 121–136, 2006.

[53] H Yin, D Song, M Egele, C Kruegel, and E Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. Conf. on Computer and Communications Security*, pages 116–127, October 2007.