

1-1-2016

User-Centric Power Management For Mobile Operating Systems

Hui Chen

Wayne State University,

Follow this and additional works at: http://digitalcommons.wayne.edu/oa_dissertations



Part of the [Computer Sciences Commons](#)

Recommended Citation

Chen, Hui, "User-Centric Power Management For Mobile Operating Systems" (2016). *Wayne State University Dissertations*. Paper 1393.

This Open Access Dissertation is brought to you for free and open access by DigitalCommons@WayneState. It has been accepted for inclusion in Wayne State University Dissertations by an authorized administrator of DigitalCommons@WayneState.

USER-CENTRIC POWER MANAGEMENT FOR MOBILE OPERATING SYSTEMS

by

HUI CHEN

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2015

MAJOR: COMPUTER SCIENCE

Approved by:

Advisor

Date

DEDICATION

To my beloved family.

ACKNOWLEDGMENTS

In 2009, I made one of the most important decision in my life and started to pursue my Ph.D. degree at Wayne State University. Studying abroad is always tough. But the people whom I had worked with during the last five years makes this part of experience enjoyable and unforgettable.

First, I appreciate the invaluable support from my advisor, Prof. Weisong Shi. With his encouragement, I started to work on power profiling and power-aware system design. In 2010, we published our first power profiling tool - pTop. That was a good start, we still occasionally receive emails from users of different countries. During my Ph.D. program, he supplied visionary guides to my research, and also provided much support to me when making my experiments. His guidance is always forthcoming. He teaches me to think big and prevents me from lost in details. From him, I learn a lot about how to thinking problems and how to express my ideas to people. Besides, he is also kind and generous to me in my daily life.

I am also indebted to the other committee members: Prof. Brockmeyer, Prof. Schwiebert, Prof. Fisher and Prof. Wang. They serve as my committee members and give valuable suggestions on my prospectus and my dissertation. From them, I learn what a professional researcher should concentrate on.

Also, I am also grateful to all my colleagues in the MIST Lab and LAST group. We usually discuss our research in the laboratory, their ideas helped me to view my research from different aspects. Huizi helped me to measure the power consumption of applications and assisted me to coordinate with volunteers during the data collecting period of my research. I am also thankful to the volunteers who agreed to do the experiment of user behavior collecting.

Last but not least, I'd like to thank my parents and my wife especially for encouraging me to pursue my Ph.D. degree. My wife sacrifices a lot to support me.

TABLE OF CONTENTS

DEDICATION	ii
ACKNOWLEDGMENTS	iii
TABLE OF CONTENTS	iv
LIST OF TABLES	ix
LIST OF FIGURES	x
CHAPTER 1 INTRODUCTION	1
1.1 Motivations	1
1.2 Objectives	3
1.3 Our Approaches	4
1.3.1 User Behavior Sensing and Analysis	4
1.3.2 Energy-efficient Location Service	5
1.3.3 Application Activity Sieving and Scheduling	5
1.3.4 Wakelock Filtering	6
1.4 Contributions	6
1.5 Outline	7
CHAPTER 2 BACKGROUND	10
2.1 Where Does Power Go in Mobile Devices?	10
2.1.1 Device Power Consumption	10
2.2 Problems of Traditional Power Management System	17
2.3 Design Rules of User-Centric Power Management Systems	18
2.3.1 Commonly Used Energy-Saving Techniques	20
2.4 Summary	22
CHAPTER 3 POWER PROFILING WITH PTOPI	24
3.1 Power Profiling	24
3.2 Methodology	25

3.2.1	Experiment Platform	25
3.2.2	Benchmarks	25
3.2.3	Experiment Method	26
3.3	Power Models	27
3.3.1	CPU	27
3.3.2	Radio	30
3.3.3	Display	33
3.3.4	WiFi	34
3.4	Related Work	35
3.5	Summary	37
CHAPTER 4	USER BEHAVIOR ANALYSIS	38
4.1	Introduction	38
4.2	User Behavior Sensing	39
4.2.1	Method	39
4.3	Pattern Analysis	40
4.3.1	User-App Interaction	40
4.3.2	Interest Point	41
4.3.3	Active Period	43
4.3.4	Usage Pattern	47
4.4	Related Work	49
4.5	Summary	51
CHAPTER 5	USER-CENTRIC POWER MANAGEMENT SYSTEM	52
5.1	Overview	52
5.2	Background	54
5.2.1	Characteristics of User Behavior	54
5.2.2	Characteristics of Mobile Devices and Applications	55

5.3	System Design	56
5.3.1	Architecture	57
5.3.2	User Behavior Monitor	57
5.3.3	Usage Pattern Analyzer	60
5.3.4	UPS Power Manager	61
5.3.5	User-Centric Application Sieving and Scheduling	63
5.4	Implementation	66
5.4.1	UPS Power Manager Service	66
5.4.2	User Behavior Monitoring	67
5.4.3	User Behavior Analyzer	68
5.4.4	UCASS	68
5.5	Experiment Method	69
5.5.1	Experiment Platform	69
5.5.2	Data Collecting	70
5.5.3	Trace-driven Simulation	70
5.5.4	Simulator	71
5.6	Evaluation	72
5.6.1	Power Consumption Analysis	73
5.7	Related Work	74
5.7.1	Power Management through Controlling Application States	74
5.7.2	User-Centric System Design	76
5.8	Summary	77
CHAPTER 6	LOCALLITE : AN ENERGY-EFFICIENT LOCATION PROVIDER	79
6.1	Introduction	79
6.2	Background	80
6.2.1	Location Service	80

6.2.2	Usage of Location Service	81
6.2.3	Location Requests Analysis	82
6.3	LocalLite	84
6.3.1	Design Considerations	84
6.3.2	Location Caching	85
6.3.3	Location Retrieving	87
6.4	Implementation	88
6.5	Evaluation	89
6.5.1	Energy Consumption Analysis	89
6.6	Related Work	91
6.7	Summary	92
 CHAPTER 7 WAKEFILTER : MAKE ANDROID FALL ASLEEP THROUGH		
	WAKELOCK FILTERING	93
7.1	Introduction	93
7.2	Wakelock Usage Analysis	95
7.3	How Long Devices Are Blocked by Wakelocks?	95
7.3.1	Which Applications Blocked Device to Sleep?	96
7.4	System Design	97
7.4.1	Wakelock Categorization	98
7.4.2	Assign Priority to Application Requested Wakelocks	99
7.4.3	Decide When the Device Goes to Sleep	101
7.5	Implementation	103
7.5.1	Energy Adaptation Support	103
7.5.2	Prioritized Wakelock Support	104
7.5.3	Wakelock Filtering	105
7.6	Evaluation	106

7.6.1	Method	106
7.6.2	Experiment Results Analysis	107
7.7	Related Work	111
7.8	Summary	112
CHAPTER 8	CONCLUSION	114
8.1	Conclusion	114
CHAPTER 9	FUTURE WORK	117
9.1	Future Work	117
APPENDIX	119
A.1	User Behavior Analysis Algorithms	119
A.1.1	Interest Point Analysis Algorithm	119
A.1.2	Active Period Analysis Algorithm	120
A.1.3	Usage Pattern Analysis Algorithm	125
REFERENCES	127
ABSTRACT	142
AUTOBIOGRAPHICAL STATEMENT	144

LIST OF TABLES

3.1	The specification of Nexus 4.	25
3.2	The micro benchmarks we used to stress hardware components.	27
4.1	The usage pattern of user U1.	49
5.1	The average power and the length of the experiment period of users.	74
5.2	The power consumption of non-system applications of user 14 in Scenario 1 and 2.	77
6.1	The location providers used by several popular Android applications.	83
7.1	The available types of wakelocks in Android, and the corresponding status of hardware components when different type of wakelocks are held.	94
7.2	The number of wakelocks requested by applications and the average wakelock length. The data is based on the device usage trace of user 6.	97
7.3	The relationship between adaptation level and wakelock priority.	98
7.4	The wakelock creators in the Android system.	99

LIST OF FIGURES

1.1	The overview of our approaches.	5
2.1	The percentage of time that the device was active during the data collecting period of the 14 users.	11
2.2	The average charging interval of all the 14 users during the data collecting period.	12
2.3	The sampled device power of user 5 during the whole data collecting period.	13
2.4	The screen state and charging state of user 5 during the whole data collecting period.	13
2.5	The percentage of CPU time consumed by applications when the device was active. The data is based on the usage trace of user 5.	15
2.6	The percentage of CPU time consumed by non-system applications and foreground applications in the active period of the devices.	15
2.7	The device power consumption of user 5 during part of the idle period. . . .	16
2.8	The percentage of idle power consumption of users.	16
2.9	The percentage of CPU time consumed by applications when device was idle.	17
3.1	The power measurement platform.	26
3.2	CPU utilization VS Power.	28
3.3	The power of the device and CPU frequency when we executed different benchmarks. Screen was on in the experiment.	29
3.4	The device power when turned on radio (without data module) and then made a phone call.	31
3.5	The device power and throughput when we uploaded 5 megabytes and downloaded 5 megabytes of data. The carrier is AT&T and network type is HSPA+.	31
3.6	The power of different colors with increasing of brightness.	33
3.7	The power of the device in WiFi scanning state. WiFi was set to scan only mode.	35

3.8	The system power and throughput when increasing upload message size from 5KB to 1M.	35
3.9	The Wi-Fi power and packet rate relationship when uploading data.	36
4.1	The average number of user-app interactions per day of all the users (based the data monitored in the first stage).	41
4.2	The time sequence of the user-app interactions of user U10 at different locations.	42
4.3	The time sequence of user-app interactions user U1 at different locations.	42
4.4	The number of locations and interest points of the users (based the data monitored in the first stage).	44
4.5	The summation of the density of the interest points (based the data monitored in the first stage).	44
4.6	The density of the time slices of user U3 and the corresponding clusters.	46
4.7	The density of the time slices of user U8 and the corresponding clusters.	47
4.8	The active periods of all the users (based the data monitored in the first stage).	48
5.1	The architecture of the UPS system.	58
5.2	The time windows of the three steps in the data processing procedure of the UPS system.	58
5.3	The flowchart of the UCASS strategy.	65
5.4	The architecture of the trace-driven simulator.	72
5.5	The percentage of energy saving when the UCASS strategy is used.	75
5.6	The extended battery life when the UCASS strategy is used.	76
6.1	The time serials of the usage of location service in about 5 days (passive requests are not included).	82
6.2	The number of location requests of users.	83
6.3	The percentage of application generated location requests of user 10.	84
6.4	The percentage of energy saved in Scenario 2.	90
6.5	The battery life extended per day in Scenario 2.	90

7.1	The percentage of time that the devices were blocked by wakelocks in the idle state.	95
7.2	The percentage of time that the device was blocked by each application during the data collecting of user 6.	98
7.3	The power measuring platform with NI cDAQ-9174.	106
7.4	The measured power consumption of Nexus 4 after the display was automatically turned off. The adaptation level of the system is 3. No third-party applications were installed on the device.	108
7.5	The measured power consumption of Nexus 4 after the display was automatically turned off. The adaptation level of the system is 1. No third-party applications were installed on the device.	109
7.6	The measured power consumption of Nexus 4 after the display was automatically turned off. The adaptation level of the system is 3. A group of third-party applications were installed on the device and ran in the background.	109
7.7	The measured power consumption of Nexus 4 after the display was automatically turned off. The adaptation level of the system is 1. A group of third-party applications were installed on the device and ran in the background.	110
7.8	Percent of time the devices were blocked by wakelocks per day when WakeFilter was enable and the adaptation level was 3 and 1.	110
7.9	Battery life saved per day when WakeFilter was enabled and the adaptation level is 1.	111

CHAPTER 1

INTRODUCTION

From early pocket-sized devices to modern mobile devices, battery life is a critical issue to address when designing a new mobile system. If this problem is not properly treated, it will significantly influence both the functionality and user experience of mobile devices. Generally speaking, the battery life of modern mobile devices is inadequate because the power consumption of hardware devices increases with performance. Moreover, existing power management systems cannot work effectively to restrict the redundant application activities (computation tasks). In this dissertation, we designed a user-centric power management system to dynamically customize the energy-saving strategies based on user behavior. We designed the *UCASS* strategy to save the energy wasted by unnecessary application activities. Furthermore, we optimized the energy consumption of location service and wakelock mechanism.

1.1 Motivations

The battery life of modern mobile devices is inadequate. Most users are not satisfied with the battery life of their smartphones. From the device usage traces of 14 users, we found that users charged the experiment devices (Nexus 4) about 1.7 times per day. The poor battery life significantly impacts user experience. Sometimes we cannot play our favorite games on the smartphones as long as we expected. Many reasons cause the poor battery life of modern mobile devices. Even though the power consumption was globally analyzed by previous publications [1, 2, 3, 4, 5, 6, 7, 8] and many optimization methods [3, 4, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28] were proposed to optimize battery life, we still cannot expect the battery life problem be solved economically soon. All aspects of operating systems must be revisited for energy-efficiency [29].

Different with old-styled mobile devices, modern mobile devices have more high-performance hardware components, which are also much more power-hungry. Besides, more third-party mobile applications, which were developed in different quality, are available and installed by

users. The power management systems of modern mobile operating systems can hardly maintain a satisfied battery life because of these new characteristics. The battery drain is mainly because existing power management systems were not functioning effectively. Usually, these power management systems were implemented based on the ACPI (advanced configuration and power interface) specification [30, 31, 32, 33]. Application activities frequently block device components to work in low-power mode. Background applications generate most of these activities. In general, most background tasks are unnecessary, redundant and even unconscious to users.

Among these application activities, many of them are related to the usage of power-hungry system services, such as location service. The information supplied by some system services is usually consistent in the same context. For example, our analysis result and several previous publications [34, 35] all found user behavior is location-dependent, and many location requests are, in fact, duplicated. These system services should be optimized to eliminate the battery drain caused by repeated usage.

Some power optimization techniques introduced in new-generation mobile operating systems also fail to decrease the power consumption of applications. For example, many previous works [36, 37, 38, 39, 40] found that wakelocks were not correctly used by some applications. This phenomenon causes the opportunistic suspending mechanism fails to function properly. The opportunistic suspending technique [41, 42] was introduced to suspend the entire device to memory when there are no meaningful tasks to execute. Otherwise, the whole system will stay in the active state. In the suspend-to-memory state, only memory and several components that monitor wake-up events are always active.

In summary, the work of this dissertation has the following motivations:

1. The battery life of modern mobile devices is inadequate. Many aspects of mobile operating systems should be revisited for energy-efficiency.
2. Existing power management systems cannot work effectively to save battery energy

wasted by redundant application activities. New power management systems are strongly needed to restrict unimportant application activities and decrease the wasted battery energy.

3. Some system services are aggressively used by applications. Many of these requests are redundant, and can be optimized to save battery energy.
4. Some mechanisms proposed in the new-generation mobile operating systems also fail to work efficiently because the misuse and abuse in applications. We should make them more energy-efficiency.

1.2 Objectives

Based on the previous observations of the current power management systems, we aim to design a new power management system to dynamically control application status, such that the energy used by unimportant application activities can be saved. Moreover, we want to make the location service more energy-efficient and make the opportunistic suspending technique functions as expected. Through these power-optimization work, we finally want to extend the battery life of mobile devices.

Background application activities is one of the primary sources of battery energy drain. From our analysis in Chapter 2, we found that most of these activities are unimportant, unnecessary or even unknown to the user. To save battery energy, we need to restrict these application activities. Both EcoSystem [43] and Cinder [44] tried to control the power consumption of applications accurately through managing battery energy as one kind of system resources. These methods assume the estimated application power represents applications' real requirement. However, it is neither accurate nor reasonable. Moreover, we also noticed that new generation mobile operating systems use energy adaptation, task grouping and computation techniques to dynamically change application status. For example, the latest iOS operating system uses the energy adaptation technique to extend battery life. When the battery level is lower than

20%, the system can work in low-power mode, which alters system behavior, such as visual effects, and application behavior, such as background application refresh, to save battery energy. However, this kind of uniform energy-saving strategies cannot satisfy all the users. Limiting application activities should not influence important applications. Otherwise, user experience will be significantly impacted. We need to design a new application-level power management system that can balance user experience and battery life.

The other objective of this dissertation is to optimize the power consumption of location service and probabilize location characterized real-time user behavior sensing. We found applications aggressively use location service. These applications either request locations too frequently (most requests were repeated), or fail to release the location service timely, or request locations that beyond their real accuracy requirement. We plan to propose a new location provider to eliminate duplicated location requests.

Finally, we plan to optimize the opportunistic suspending mechanism of the Android system and also restrict wakelock abuse. Our analysis result, as well as previous publications [36, 37, 38, 39, 40], found wakelocks were commonly misused and abused. The phenomenon usually makes the opportunistic suspending mechanism fails to suspend the device to low-power mode. Some applications, often, fail to release wakelocks timely, or never release wakelocks.

1.3 Our Approaches

As illustrated in Figure 1.1, our approach includes four parts. In this section, we generally describe them.

1.3.1 User Behavior Sensing and Analysis

To optimize the power consumption of applications without influencing user experience too much, we need to understand the preferences of users and customize the energy-saving strategies for them. In this dissertation, we define user behavior as the interactions between users and applications. We use statistical and data mining algorithms to analyze the usage

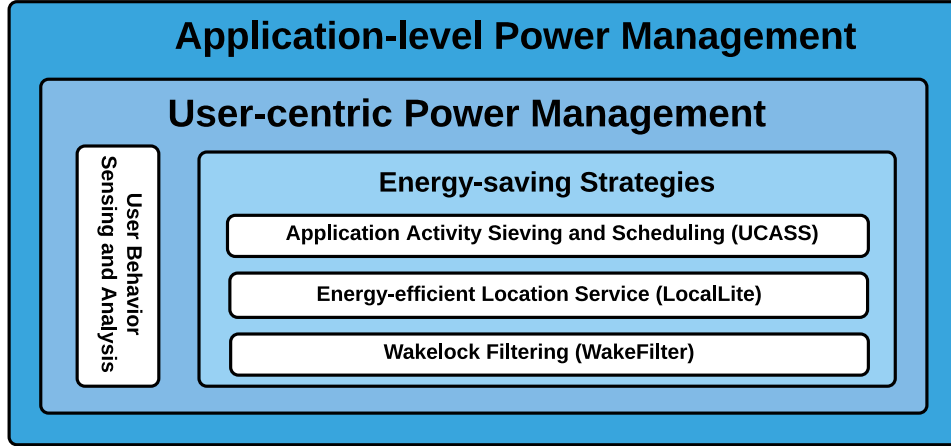


Figure 1.1: The overview of our approaches.

pattern of users from the sensed user behavior. The usage pattern includes a group of pattern items, each of which shows how frequently the user used an application at the given period and location. A pattern item also means the user will probably use the same application at the specified location and time in the near future, thus we can use usage pattern to distinguish which application activity is important.

1.3.2 Energy-efficient Location Service

User behavior is location-dependent. To decrease the power consumption of location service and probabilize location-marked user behavior sensing, we use location caching to solve the problem. In our user behavior analysis result, we found users frequently used the devices at some dedicated locations; most location requests were, in fact, duplicated. Based on this phenomenon, we combine location with the context, which is marked by the wireless access point, and cache the received location as well as the information of the wireless access point to data storage. When an application requests location in the same context, the cached location is returned without start the hardware component to pinpoint the location.

1.3.3 Application Activity Sieving and Scheduling

Application activities frequently cause hardware components work in high-power mode. Most background application activities are redundant to the user and can be optimized to de-

crease the power consumption. To restrict background application activities without influence user experience too much, we use user's usage pattern to distinguish critical application activities from regular application activities. If an application has a pattern item in the current context (a combination of time and location), we can say it is an important application. Thus, the activity of this application should not be altered.

We use the task grouping technique to reschedule the tasks of background applications. Based on usage pattern, these tasks may be scheduled to execute immediately, in the active period or charging period of the device. In this way, we can significantly decrease the battery drain. In the charging state, we see application power consumption as zero, a similar method was also used in [45]. When the system executes tasks in groups, the tail power consumption of hardware components can be saved [44]. The usage pattern is also helpful to design other energy-aware strategies. To facilitate researchers to concentrate on the design of policies, we propose the *UPS* system, which is a framework that bridges user behavior and energy-saving strategies. It senses user behavior, analyzes the usage pattern of users, and listens a group of system events to trigger energy-saving strategies.

1.3.4 Wakelock Filtering

A wakelock or suspend blocker is used to guarantee the execution of an important task. Developers, however, rarely considered the energy consumption of applications, and applications often failed to release wakelocks on time. Furthermore, they use wakelocks aggressively in applications. To solve the problem, we applied the energy adaptation technique to the power management system. In the energy adaptation state (when battery level is low), applications and system services should change their behaviors accordingly to save energy [46, 47]. We utilize usage pattern to distinguish critical applications and unimportant applications, and assign a different priority to their wakelocks. In the energy adaptation state, we ignore low-prioritized wakelocks such that fewer wakelocks block the device to sleep.

1.4 Contributions

In this dissertation, we claim that we have the following contributions:

1. We found the user behavior of users is both time-dependent and location-dependent through the analysis of 14 users' user behavior. In the contexts that marked by time and location, users are inclined to use a particular group of applications. These observations are helpful for designing user-centric systems.
2. Based on the analysis result of device power consumption, we found background application activities consumed a considerable amount of battery energy in both the idle state and active state of the device. Most of these background tasks are unimportant, unnecessary or even unknown to users. We designed the *UCASS* energy-saving strategy to optimize this situation. Our experiment result shows the *UCASS* strategy can save about 25.62 percent of the energy consumed by background application activities, and extends battery life for about 25 minutes per day on average.
3. From the analysis result, we also observed that many location requests were redundant because applications requested them in the same context. We proposed the *LocalLite* location provider to eliminate the redundant location requests. The experiment results show that *LocalLite* can efficiently reduce 98.51 percent of the energy consumed by location requests. It also enables location-marked consecutive user behavior sensing.
4. Finally, we found that applications aggressively use wakelocks. The opportunistic suspending mechanism of Android system cannot work as expected because application generated wakelocks frequently blocked the device to sleep. We proposed the *WakeFilter* strategy to filter low-priority wakelocks in energy adaptation mode, such that devices can change to the deep-sleeping state. Our experiment result shows *WakeFilter* decrease about 65.55% of device-blocked time, and it can extend battery life for about 1.58 hours per day in the best case.

1.5 Outline

The following chapters of the dissertation is organized as follows:

In Chapter 2, we analyze the device usage traces of 14 users. We try to find out how users interact with devices; how users interact with applications; and how battery energy was consumed in both the device idle and active state; and how background applications consume energy. We found background application activities consume a considerable amount of battery energy. These background activities are either unnecessary or unimportant to the user. Besides, we analyze the existing power management systems and discuss the limitation of these systems. Finally, we summarize commonly used techniques to optimize application power consumptions. These analysis results motivate us to propose user-centric power management systems in Chapter 5.

Chapter 3 describes the power models and discusses our experience in designing and calibrating power models. We implemented these power models in the simulator described in Chapter 5, and use it to simulate application power consumptions in different scenarios.

Chapter 4 describes the statistical and data mining algorithms that we proposed to analyze the interest points, active periods and usage patterns. With these algorithms, we analyzed the usage traces of users. The experiment result shows most users interact with their devices more frequent at some specific locations (interest points) and periods (active period). This observation proves user behavior is both time-dependent and location-dependent. Based on these two contextual elements, we analyzed the usage pattern of users. The usage pattern is used in the design of the *UPS* system, described in Chapter 5, and the *WakeFilter* energy-saving strategy, described in Chapter 7. Also, the design of *LocalLite*, described in Chapter 6, is based on our observations of interest points.

In Chapter 5, we describe the design and implementation of the *UPS* system, which is an open framework that bridges user behavior with energy-saving strategies. Based on the framework, we can quickly develop new energy-saving strategies. Also, we designed a default

energy-saving strategy named *UCASS*, which decreases battery energy consumed by background applications through rescheduling and grouping tasks to execute in device charging or device active state. It utilizes usage pattern to make scheduling decisions. Based on the *UPS* power management framework, we also optimized the power consumption of location service in Chapter 6 and optimized the wakelock mechanism of Android in Chapter 7 to make the device get more chances to sleep.

Chapter 6 describes the design and implementation of *LocalLite*, which is an energy-efficient location provider. It is designed based on the observation that we found in Chapter 4. From our analysis, we found many wakelock requests generated in the same location are redundant and thus should be optimized for energy-efficiency. We cache location with the contextual information. When the user requests location again in the same context, the cached location is returned.

In Chapter 7, we found wakelocks are abused and misused by applications in mobile systems. To solve the problem, we propose *WakeFilter*, which filters low-priority wakelocks in the energy adaptation state to make the devices work more in the suspend-to-memory state. The priority of wakelocks is assigned based on user behavior that supplied by the *UPS* system we designed in Chapter 5.

We conclude the work of this dissertation in Chapter 8. Finally, we discuss future works about how to improve the *UPS* system, designed in Chapter 8.

CHAPTER 2

BACKGROUND

Many reasons cause the poor battery life of mobile devices. In this section, we describe the background information, such as power dissipation, power management systems and power optimization techniques, of this problem.

2.1 Where Does Power Go in Mobile Devices?

When we talk about where does power go in mobile devices, the first impression to this question is hardware components consume battery energy. Many previous work [3, 11, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57] analyzed the power consumption of various hardware components. It is true that hardware components draw power, but it is not completely true because application activities make hardware components active. Dynamically controlling the power state of hardwares is a passive way of power management. This method cannot effectively save battery energy consumption when applications generate too many redundant activities. In this section, we analyze how mobile devices consume battery energy based on the device power consumption and application information we monitored from 14 volunteers.

2.1.1 Device Power Consumption

We modified the Android system to record user-app interactions, system events, application activities, wakelock request information, location service usage, and real-time device power. The experiment platforms (Nexus 4), which were flashed with the modified system image, were given to 14 users (their age ranges from 22 to 35) to monitor these data during their daily usage. Half these 14 users are graduate students, the other half of users are employees that working on technical jobs; 5 of them are female users. Each user used the device for about a week. Since the active time (screen on) caused by applications, such as the desk clock, that can wake up the device are very short, we see all the active time as caused by user operations in the analysis. In this dissertation, we define the active state of the device as when the screen is on, and define the idle state of the device as when the screen is off.

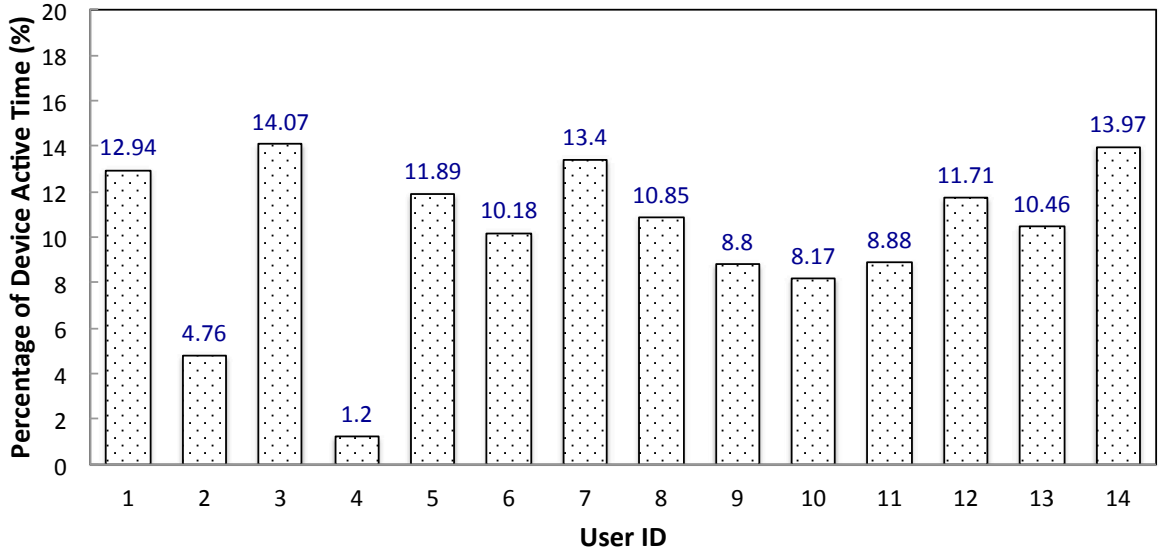


Figure 2.1: The percentage of time that the device was active during the data collecting period of the 14 users.

Device Active and Charging

We first analyzed the *POWER_CONNECTED* and *POWER_DISCONNECTED* events (device charging state) and the *SCREEN_ON* and *SCREEN_OFF* events. From Figure 2.1, we can see that the device active time of all users is less than 15% during the experiment. The devices were active for only about 10.09% of the time on average. However, the average charging interval was about 15.31 hours, shown as Figure 2.2. Users charge the devices about 11.94 times per week on average, which means users had to charge the devices more than once a day. This observation shows the battery life of our experiment devices is very poor. Where does power go in the mobile devices? To answer this question, we need to analyze both the active period (screen on) and idle period (screen off) power consumption of the device.

Active State Power Consumption

Figure 2.3 shows the device power dissipation of user 5 during the whole experiment period, and Figure 2.4 shows how the charging state and screen state changed during same period.

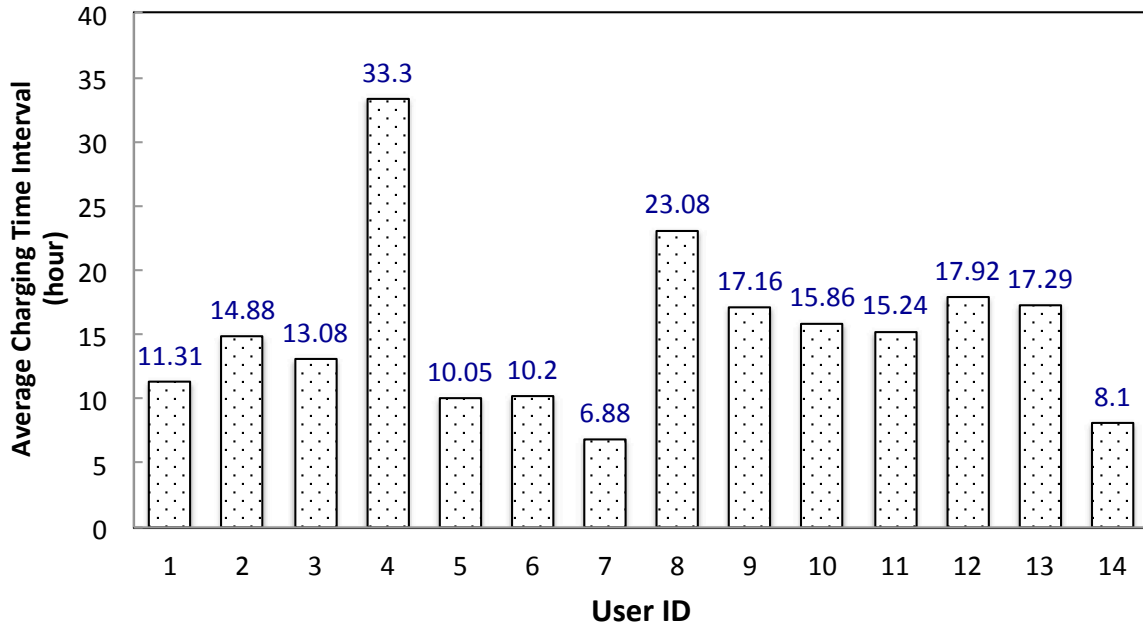


Figure 2.2: The average charging interval of all the 14 users during the data collecting period.

Because the device power were calculated from sampled battery voltage and current, the power value is negative (shown as 0 milliwatts in the figure) when the device was charging. In Figure 2.3, the high power dissipation was generated when the device was active. On the one hand, hardware components that work in high-power mode caused the power dissipation. On the other hand, application activities also cannot be ignored.

Even though the power consumption was very high in the active state, the device stayed in this state for only about 11.89 percent of the time. The device became active for about 18.62 times per day, and each active period lasts for about 9.19 minutes on average. In some situations, the device power dissipation was more than 3000 milliwatts.

Which applications caused the power consumption when the device is active? To answer this question, we analyzed the CPU time consumed by applications. When the device is active, system applications (root, android, media server, sensors and systemui) are the main consumers of CPU time. Their CPU time accounts for about 62.7 percent, shown as Figure 2.5, and non-system applications used about 37.3 percent. However, we know that some system applications

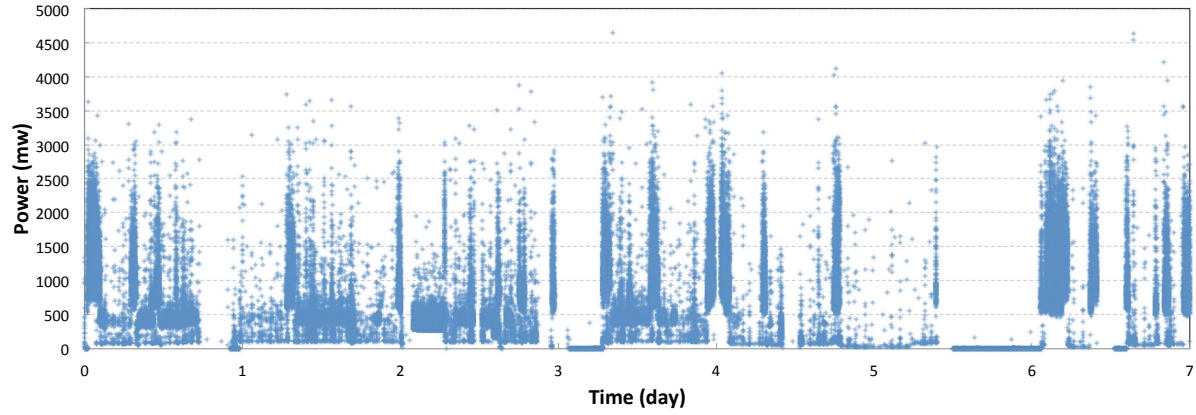


Figure 2.3: The sampled device power of user 5 during the whole data collecting period.

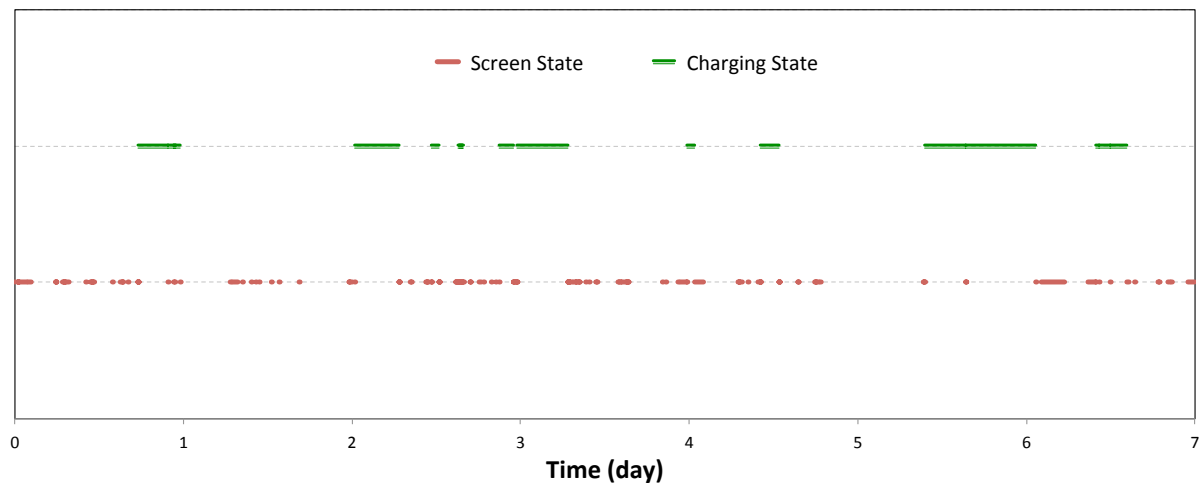


Figure 2.4: The screen state and charging state of user 5 during the whole data collecting period.

like *android* are a group of system services that support the execution of non-system applications. If the activities generated by non-system applications decrease, the CPU time consumed by system applications will also decrease accordingly. In fact, only 20.57% CPU time was directly used by foreground applications, and background application generated activities used as much as 16.73% CPU time.

For the other users, the situation is similar, as shown of Figure 2.6. Foreground applications used less than 30 percent of CPU time. In this figure, the experiment result of user 8 and user 11 shows that non-system applications only used a very small amount of CPU time. After careful analysis, we found the *root* system application (uid is 0) consumes most of CPU time. In the data-monitoring period, the root application handles low-level I/O operations (read system files and write logs). A large amount of CPU time was consumed by non-system applications that executed in the background. Based on our knowledge, most background applications' activities are not useful, unnecessary or even unknown to users. Thus, the battery energy consumed by them and the corresponding system services were wasted.

Idle State Power Consumption

From Figure 2.3, we can see that the idle period of the device was much longer than the active time. The idle state power dissipation (power consumption when the screen is off) of the device was much lower than the active power dissipation (power consumption when the screen is on). In some idle period, as shown of Figure 2.7, the device consumed a large amount of power to process background application activities. In this period, the average power of the device was about 431.21 milliwatts. From this figure, we can see that the device was not really in sleeping state but periodically waked up. All the users' idle state power consumption is more than 15%, shown as Figure 2.8.

Again, we use user 5 as the example to analyze application power consumption in the idle state. From figure 2.9, we found that about 69.27% of CPU time is consumed by system appli-

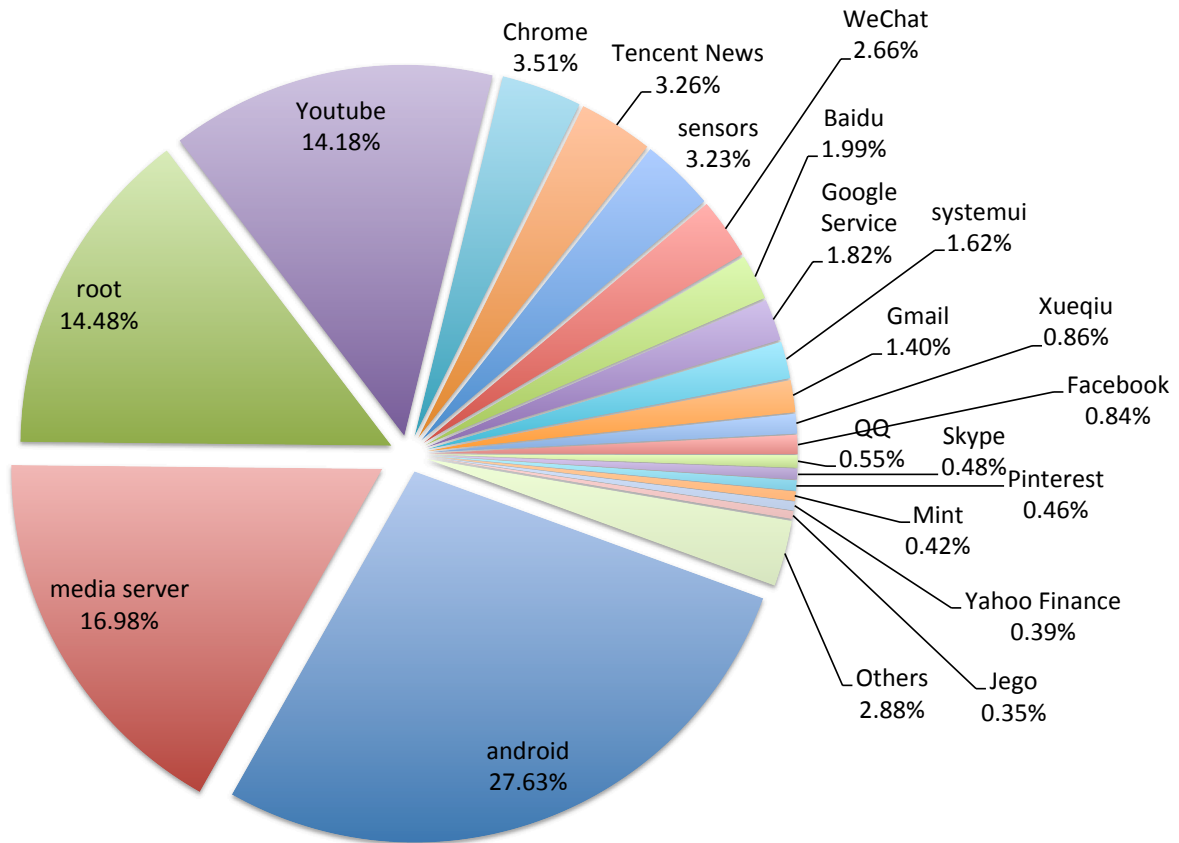


Figure 2.5: The percentage of CPU time consumed by applications when the device was active. The data is based on the usage trace of user 5.

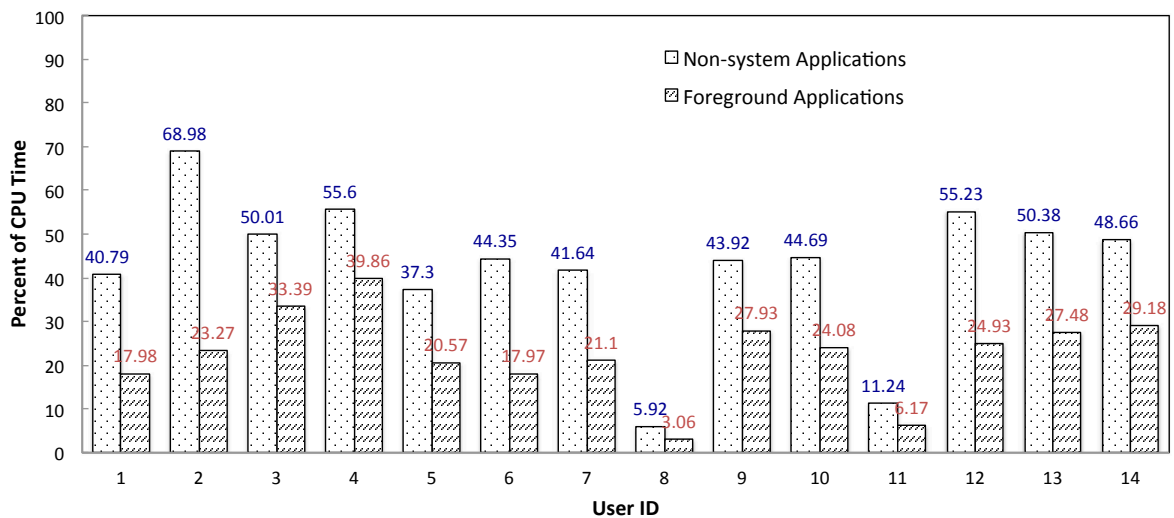


Figure 2.6: The percentage of CPU time consumed by non-system applications and foreground applications in the active period of the devices.

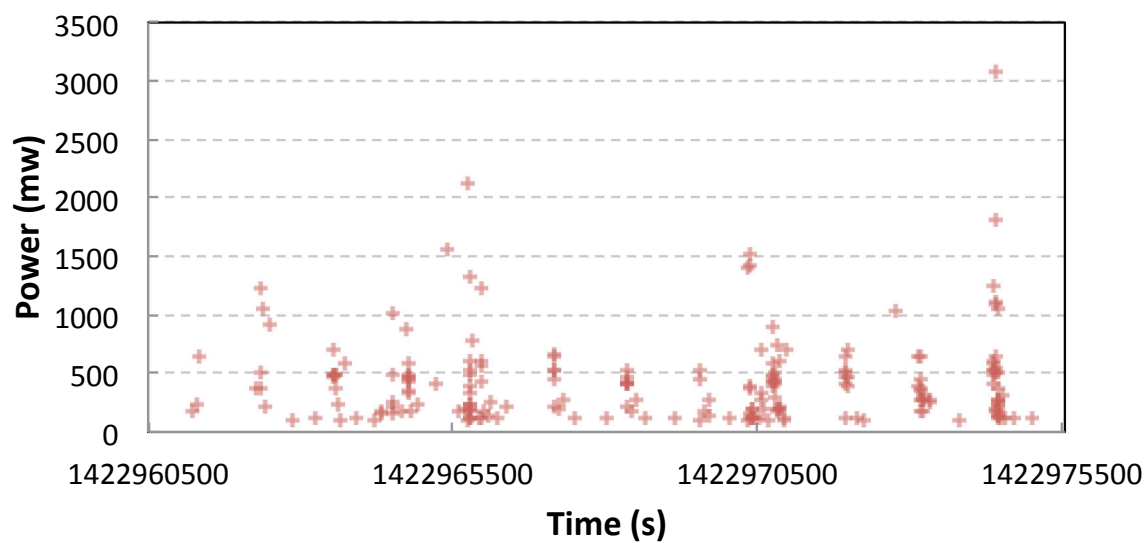


Figure 2.7: The device power consumption of user 5 during part of the idle period.

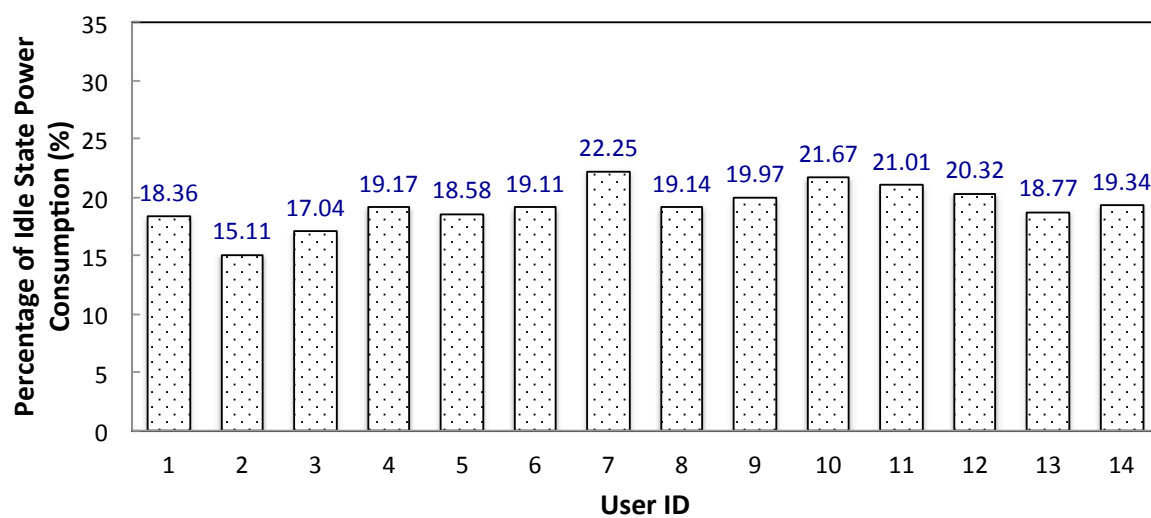


Figure 2.8: The percentage of idle power consumption of users.

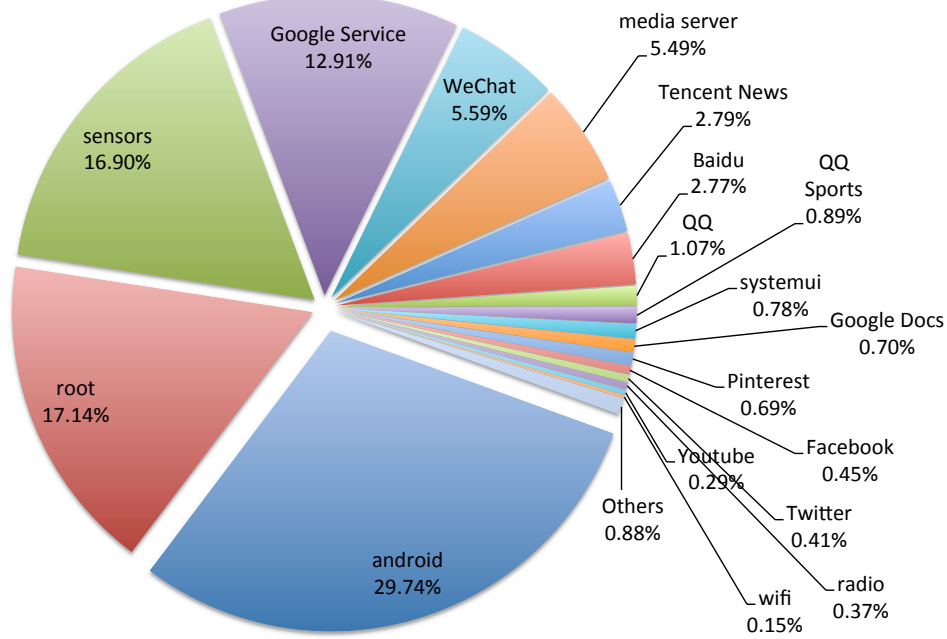


Figure 2.9: The percentage of CPU time consumed by applications when device was idle.

cations, such as android, root, sensors, media server and so on. The top five non-system applications (Google Play Service, WeChat, Tencent News, Baidu and QQ) only account for about 25.13 percent. However, system applications' activities are generated with non-system applications. For example, media server runs with audio player or video player. Most non-system applications periodically wake up the system to synchronize with cloud services. Based on our knowledge, most of these application activities are not necessary or not required to perform frequently. We think most of the energy consumed by background activities are wasted.

2.2 Problems of Traditional Power Management System

No matter, the device is in the idle state or active state, non-system applications run in the background generated plenty of activities and consume a considerable amount of battery energy. These activities are either unimportant, unnecessary or unknown to users. They are not generated by or related to the direct user operations. Previous publications [58, 59, 30, 60, 31, 61, 17, 62, 18] that tried to optimize the power management systems, which were implemented based on the ACPI specification [63], cannot effectively tackle this problem because they tried

to control the power state of hardware components. However, the redundant background activities cause these hardware components keep in the active state much longer than we expected. To make ACPI-based power management systems work effectively, we need to filter, restrict or reschedule background tasks. We name this kind of power management as application-level power management, and the traditional power management as component-level power management.

Furthermore, foreground applications should also be changed to optimize power consumption, especially when the battery level is low. For example, is playing high-resolution video necessary to users on smartphones? Because the limitation of screen size, high-resolution videos cannot greatly increase user experience. If we can control video players to decrease the resolution requirement, much less network transmission will be generated, and much less CPU time will be used to decode video. This kind of technique named energy adaptation [46, 47], which was already implemented in the latest iOS 9 operating system. When the battery level is low, the user will be notified to disable background application refresh. Also, the iOS system alters itself, such as turning off the visual effects, to save power. But, this kind of uniform energy-saving strategies can not satisfy all the users. For some users, user experience may be significantly impacted if an important application was restricted to refresh in the background. In the future, many aspects of the system and applications should be optimized to improve the energy adaptation technique.

2.3 Design Rules of User-Centric Power Management Systems

Based on these observations, we can optimize system power consumption from two directions: (1) optimize the power consumption of system services; (2) decrease application activities, so that system services also generate fewer activities. For the first direction, we need to revisit each system service of mobile operating systems and make them more energy-efficient. This part of work is directly related to the specific operating system. In this paper, we improved the power consumption of location service and wakelock usage. The second

direction is much more generic. Even though we implemented our *UPS* system in Android, the solution can easily be applied to other mobile operating systems. Following this direction, we proposed the *UPS* system and designed the *UCASS* energy-saving strategy to optimize the power consumption of mobile operating systems.

The energy-saving strategies of application-level power management directly change the behavior of applications. Thus, they must be carefully designed. Otherwise, user experience will be significantly influenced. A power management strategy that is good at energy-saving but significantly reduces the quality of user experience will not be accepted, vice versa. In this dissertation, the energy-saving strategies we introduced follow several design rules:

1. A power optimization solution should not influence the execution of foreground application. Interacting with the foreground application is the most important operation for users.
2. The system services that support the current foreground application should not be influenced by power management. Otherwise, the foreground application cannot run as normal.
3. A power-saving strategy must not restrict the main functions, such as phone call and text message, of the smartphone. No matter how we design the power management system, we should always ensure the key functions of the smartphone. Even though modern mobile phones provide various functionalities, communication is still the primary function.

To obey these rules, we need to know which application is important to the user at different situations or context. Previous publications [64, 65, 66, 35, 67, 68, 69, 34], as well as our experiment in Chapter 4, found that the user behavior of most users follows a pattern. Moreover, the usage pattern of users is both time-dependent and location-dependent. The usage pattern is represented as association rules between usage context and application. Some of the previous work [66, 35, 67, 68] already utilized user's usage pattern to assist system design. For example,

Shin *et al.* proposed a method anticipate the next application that the user will use from the user's usage pattern [35]. In this dissertation, we apply the usage pattern to design user-centric energy-saving strategies. Since we can distinguish significant applications (applications that the user probably will use soon) from normal applications in the different context, important applications' behavior will not be influenced in the design of the energy-saving strategy. Such that, we can effectively balance user experience and energy saving. We named our user-centric power management system as the *UPS* system and implemented in on the Android system. We will talk more about it Chapter 5.

2.3.1 Commonly Used Energy-Saving Techniques

Application developers rarely consider the energy-efficiency of their applications; energy-aware design may increase the complexity of the application and the cost of the development. From the system side, we can design strategies to manage application activities as well as their power consumption. In this section, we discuss three commonly used techniques for optimizing application power consumption.

Energy Adaptation

The first commonly used energy-saving technique is energy adaptation. Usually, we decides the energy adaptation state based on battery level. When the system switches to energy adaptation mode (battery level is lower than a threshold), both system and applications should dynamically change their state to save battery energy. This technique was proposed because researchers found the great potential to save battery energy through altering application behaviors. For example, downloading and playing a low-resolution video consumes much less energy than a high-resolution video. This latest iOS system [70] implemented this technique. The system turns off background application update when the battery level is lower 20%, and resumes to normal state when device power is more than 80%.

In our previous work [47], we proposed the *Anole* framework to support energy-aware de-

sign for applications. This design assumes users are willing to sacrifice part of user experience to extend battery life. In [43], the EcoSystem proposed by Zeng *et al.* also utilized the idea of energy adaptation. They assume the expected battery life is known, and then a limited amount of battery energy is allocated to applications in each epoch to guarantee the expected battery life. In the Cinder operating system [44], battery energy consumed by applications is also strictly controlled. If an application does not have sufficient allocated energy to perform an energy-intensive operation, it has to be suspended and waits until there is enough battery energy allocated to it. Anand *et al.* proposed a dynamic adaptation method to decrease the power consumption of Games [20]. They save display power consumption through exploiting the Gama function of games to dim LCD backlight.

This energy-saving technique obviously influences user experience. To use it, we need to trade carefully off between user experience and energy saving. Otherwise, it is hard for the users to accept it. To mitigate the impact on the quality of user experience, we should distinguish important applications from normal applications and guarantee the execution of important applications.

Task Grouping

Task grouping is another commonly used power-saving technique. As we know that most device components were designed based on the ACPI specification. A device component is not changed to lower power state immediately if all the tasks are processed. That's because changing the power states of components takes time. For example, after radio finishes all the data transmission, it will stay in the high-power state for a tail time T . In the tail time, it resumes transmitting data when new data transmission tasks arrive.

Based on this phenomenon, researchers proposed to group similar tasks. In Cinder [44], Roy *et al.* proposed the idea of cooperation. Applications that use the same device, such as WiFi, can work with each other to activate the device and execute their tasks together. Bal-

asubramanian *et al.* proposed TailEnd to reschedule delay-tolerant applications' network requests [51]. In the *UPS* system, we also use the task grouping technique to reschedule some background application's activities to execute together. Different with them, we use the user's usage patterns to schedule application background tasks to decrease the impact on user experience.

Computation Offloading

Even though the performance of mobile devices improves very fast, we cannot move too many computation-intensive tasks to the client-side because of the battery life problem. On the contrary, offloading part of heavy computation tasks to the server-side is more energy-efficient. Chen *et al.* proposed to offload compilation tasks to the server-side [71]. They consider communication, computation and compilation energy consumption, and automatically decide where to compile and execute a method is more energy efficient. Similarly, Cuervo *et al.* also proposed a framework named MAUI to decide which method should be offloaded to the cloud.

Liu *et al.* [21] proposed to move the location computation to the server side, in this way, only a very limited amount of raw GPS signals are needed to pinpoint the location. It reduces the sensing time of GPS signal. Chun *et al.* developed a system to support application partition and execution on cloud and local [72]. They used both the static analysis and dynamic profiling techniques to partition applications into fine-grained parts, and optimize the energy consumption for the computation. Segata *et al.* researched the power models of various network devices used for trading off computation offloading and local execution [73]. In [74], Qian *et al.* presented the Jade system to dynamically adjust offloading strategy based on workload and communication cost.

2.4 Summary

In this chapter, we analyzed the device power consumption and application power consumption in the idle state and active state. From the analysis, we found background application activities make the mobile device active frequently and consume a considerable amount of battery energy. Moreover, we discussed the problem with existing power management systems and listed the rules to design user-centric power-saving strategies. Finally, we talked about three commonly used energy-saving techniques that save energy through altering application activities.

In the next chapter, we discuss our experience in designing power models. We used these models to estimate application power consumption and utilized it in the evaluate of the energy-saving strategies we proposed.

CHAPTER 3

POWER PROFILING WITH PTOPT

Understanding the power dissipation of mobile applications is essential for developing energy-efficient applications and optimizing the energy consumption of mobile systems. In this dissertation, we use application power information to simulate the power management system and evaluate the energy-saving strategies. Because we cannot generate exactly the same user activities and fairly compare the energy consumption of different scenarios, thus we use the simulation method to do the experiment. In this chapter, we talk about the method that we used to estimate the power consumption of applications.

3.1 Power Profiling

The dynamic power that caused by software activities dominates the power consumption of modern mobile systems. This phenomenon is mainly because the global usage of dynamic voltage and frequency scaling (DVFS) and clock throttling techniques. Understanding application power consumption is not only important to investigate power issues but also critical to evaluate energy-saving strategies. Power modeling is commonly used to calculate the power consumption of hardware components, processes, and applications. In our previous research [75, 76], we constructed process-level power models, and developed the profiling tool (pTop) to estimate application power consumption. In this chapter, we revisit these power models and calibrate them for the experiment device we used in this dissertation.

To build the power models, we have to find a group of power indicators, each of which is correlated with one or several hardware components' power dissipation. For example, in some previous publications [56, 77, 55], a group of hardware performance counters (HPC) were used to build power models. In addition, a bunch of power models for different hardware components [49, 52, 54, 57] were proposed, and several power profiling applications [78, 36, 53] were developed. Similar to them, we build power models in two steps: (1) construct power models for hardware components based on the state of power indicators; (2) allocate the power

of components to applications based on the percentage of resource utilization.

3.2 Methodology

To find the suitable power indicators for each hardware component, we need to isolate the power consumption of hardware components with a group of micro-benchmark applications. Each micro-benchmark only stresses one hardware component, thus we can analyze the correlation between the measured power and the power indicators. Good power indicators are usually highly correlated with the power dissipation of hardware components. In this section, we talk about the experiment platform and method that we used to build power models.

3.2.1 Experiment Platform

As illustrated in Figure 3.1, we use Nexus 4 (flashed with Android 4.4.2 and Linux kernel 3.4.0), BK Precision programmable power supply and a laptop to set up the power measurement platform. The anode of Nexus 4 is only connected to the anode of the power supply to bypass the battery. The cathode of Nexus 4 is connected with the cathode of battery and also the power supply. In the experiment, the power supply was set to 3.8 volts, and we set the maximum current to 3 amperes. The power supply samples current four times per second, and the computer collects the sampled data. Table 3.1 lists the specification of the smartphone.

Component	Specification
OS	Android 4.4.2; kernel version 3.4.0
Chipset	Qualcomm Snapdragon APQ8064 S4 Pro
Processor	Quad-core Krait; 384 - 1512 MHz;
RAM	2G ; Dual-channel 533 MHz LPDDR2
Display	4.7 in diagonal IPS; 1280x768 px; 320 dpi
GPU	Adreno 320
Radio	Integrated 3G/4G World/multimode
Wi-Fi	Integrated digital core 802.11n (2.4/5GHz)

Table 3.1: The specification of Nexus 4.

3.2.2 Benchmarks

Since our experiment platform can only measure the total power of the device, we developed a set of micro-benchmarks, shown as Table 3.2, to stress several main hardware compo-

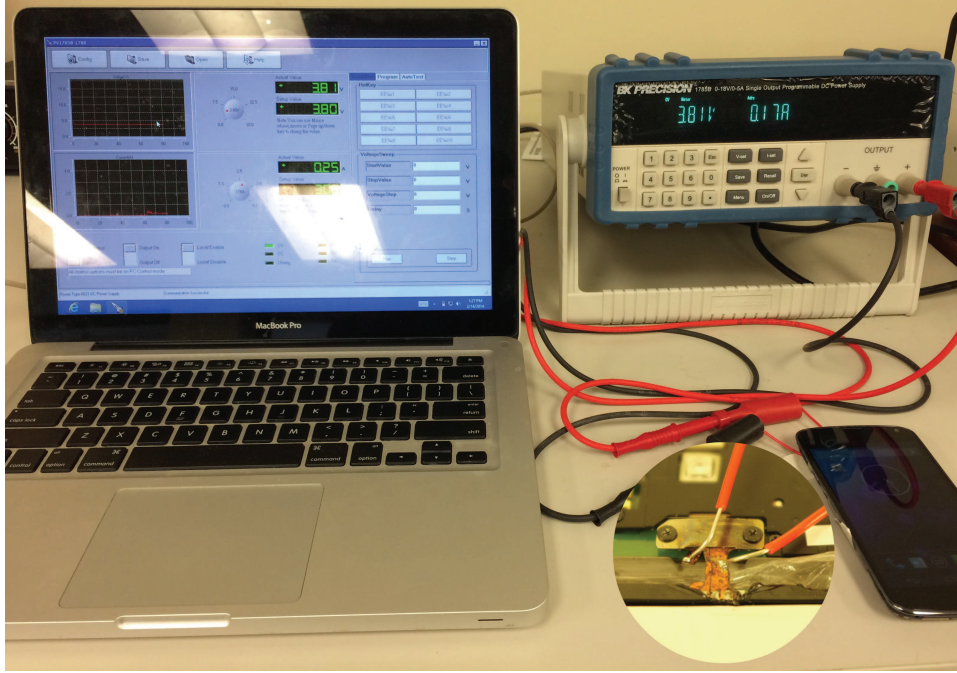


Figure 3.1: The power measurement platform.

nents. While each micro benchmark is executing, only part of the hardware components are active (CPU and memory are always active). So that, we can isolate the power consumption of each device, and use the subtraction method to calculate the power of them. For example, the *prime* benchmark, which searches for prime numbers, can be fully loaded to cache. Thus, all the active power is mainly caused by CPU.

3.2.3 Experiment Method

For each component, we did a lot of experiments to find out the suitable power indicators and the base power to calibrate its power model. Because CPU, screen and memory are always active when we run each benchmark, we built power models to estimate their powers first, and then calibrate the power models of other components. When we did the experiment for a component, all the other unrelated components were turned off. For example, when we execute 3G/4G targeted benchmarks, we turned off WiFi, GPS, Bluetooth, etc.

Benchmark	Description
Prime	Search for prime numbers.
Cache	Force to read data from the specified level of memory system.
Pi	Calculate the number π of the specified digits.
Math	Make integer and float computations.
Screen	Set the screen to different brightness level and foreground colors.
SensorHub	Read different sensors.
Socket	Upload and download the specified size of messages to socket server via cellular network or wireless network.
Location	Locating with GPS or network location providers.

Table 3.2: The micro benchmarks we used to stress hardware components.

3.3 Power Models

In this section, we discuss how to choose power indicators for several power-hungry components, which consumes around 70% of the total device power [3].

3.3.1 CPU

Previous research generally use CPU state information, such as frequency and utilization, and hardware performance counters(HPCs) to build power models. HPCs are unsuitable to be used in OS-layer services because they consume a large amount of CPU cycles.

CPU Utilization

Many previous publications use Equation 3.1 to calculate CPU utilization. However, the idle time and I/O wait time used in this equation may be wrong in the tickless kernel. When a core enters into the C3 state, the local APIC(Advanced Programmable Interrupt Controller) timers and the timer interrupt for scheduling will also be shut down. Then, wrong idle time and I/O wait time will be reported when the number of active core changes. To solve the problem, we use a different Equation 3.2, in which ΔT is the interval between two consecutive system state sampling, to calculate CPU utilization.

$$U = (\Delta T_{sys} + \Delta T_{user}) / (\Delta T_{sys} + \Delta T_{user} + \Delta T_{idle} + \Delta T_{iowait}) \quad (3.1)$$

$$U = (\Delta T_{sys} + \Delta T_{user}) / (\Delta T * CoreNumber) \quad (3.2)$$

To verify the correlation between CPU utilization and power, we run the Prime benchmark with a different number of threads. From Figure 3.2, we can clearly see the linear relationship between them. The correlation coefficient is 0.9945. Moreover, the dynamic power ranges from about 52 milliwatts to 3560 milliwatts. Partial wakelock was acquired to keep CPU active while the screen was turned off.

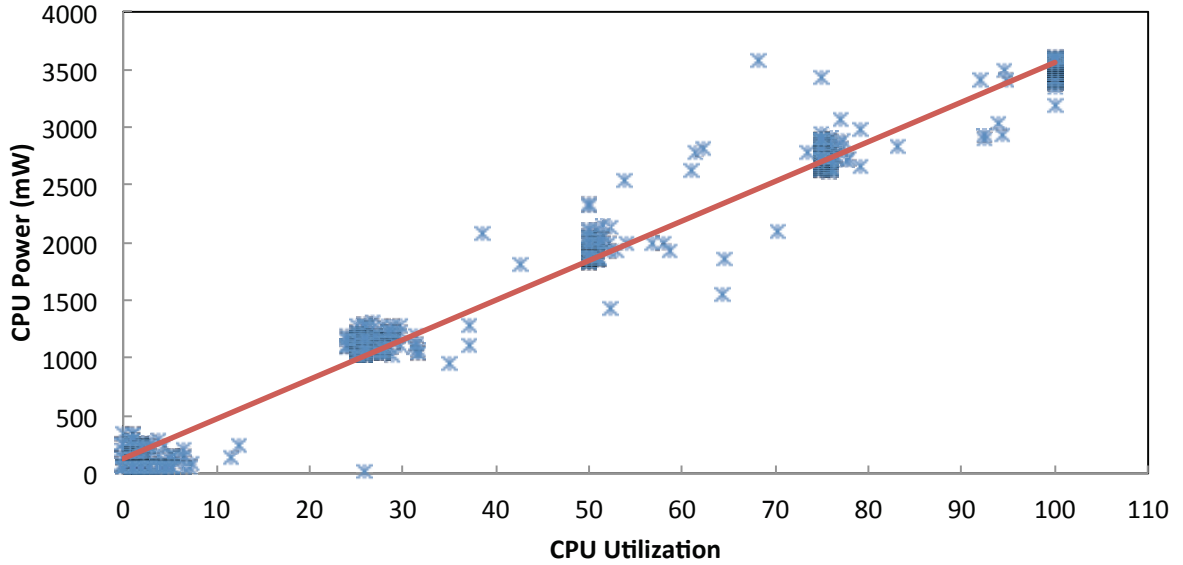


Figure 3.2: CPU utilization VS Power.

Frequency

Some of the prededing work, as well as our previous publications [76, 75], use the CPU frequency to decide the range of dynamic power. In the current mobile systems that use the

default ondemand CPU frequency governor, frequency has much less influence to power dissipation. The ondemand governor increases CPU frequency to the maximum if the workload increases a little. Then, if the workload abates, it will step back slowly. This means that CPU frequency influences power consumption only when the workload is very low. In one of the experiments, we set CPU governor to userspace¹ and gradually increase CPU frequency from the minimum to the maximum. We observed that the power only increased about 41.8 milliwatts. Figure 3.3 does not show us a direct relationship between power and frequency. The correlation coefficient between CPU frequency and power is only 0.0892 in this experiment.

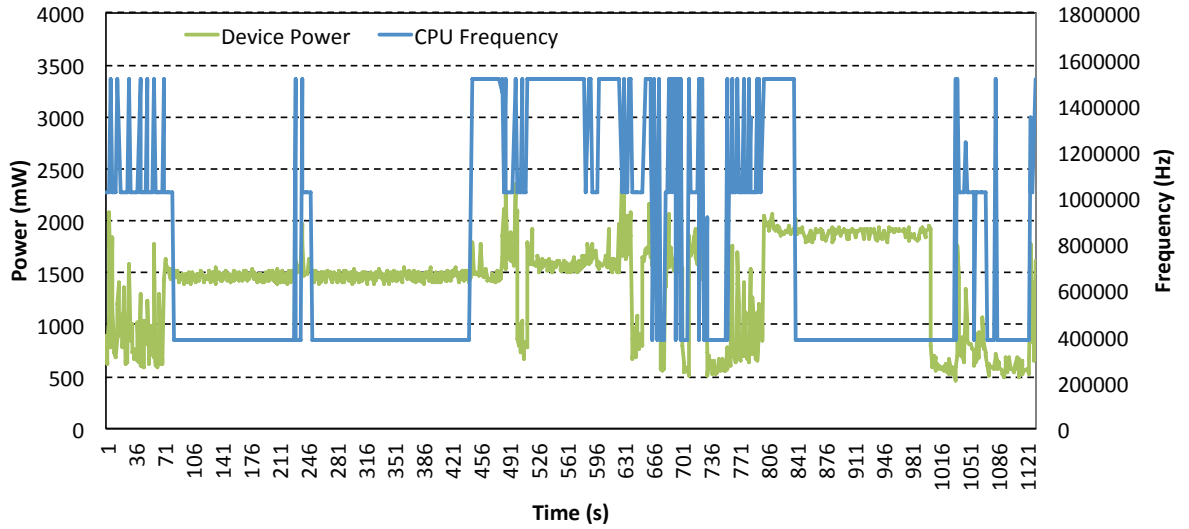


Figure 3.3: The power of the device and CPU frequency when we executed different benchmarks. Screen was on in the experiment.

Power Model

We construct CPU power model as Equation 3.3 based on the previous analysis. Then, we allocate the estimated CPU power to processes and applications based on their CPU utilization in each sampling period.

¹The userspace governor allows users to manually set up CPU frequency.

$$P_{cpu} = P_{cpu}^{static} + P_{cpu}^{max} * U \quad (3.3)$$

Low overhead is the main merit of this power model. The drawback is that it cannot accurately estimate the power when part subsystems of the processor, such as L1 and L2 cache, is heavily stressed. When we execute the cache benchmark to stress L1 cache, L2 cache, and memory, the corresponding measured CPU power is about 1444 milliwatts, 1064 milliwatts, and 912 milliwatts. However, the CPU utilization in these tests is about the same. We think this type of extreme case rarely happens and will stay with the power model in our implementation.

3.3.2 Radio

The power model of radio should consider both the voice module and the data module. In this section, we talk about the power indicators of these two modules.

Voice Module

The power of the voice module depends on the state of this module. It can work in several states, such as the call state, radio service state, SIM card state and radio signal strength. As shown in Figure 3.4, when we turn on the radio (data service is disabled), the radio service start signal scanning. After the device had registered to the carrier's network, the radio was in the idle state most of the time. Based on the difference of signal strength, the idle power of voice module changes between 4.4 milliwatts and 8.17 milliwatts. The radio is in the active state when the call state is either ringing or offhook. The significant power fluctuation when ringing is caused by CPU activities, media (play ring sound) and vibrate motor. The power model of the voice module is shown as Equation 3.3.2. We add the power of the voice module to the *Phone* application.

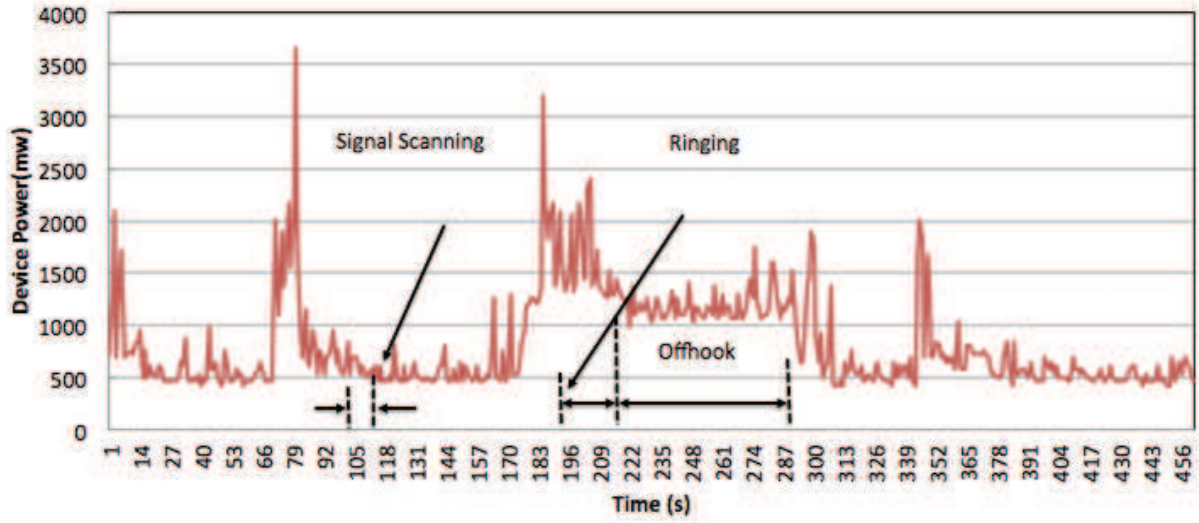


Figure 3.4: The device power when turned on radio (without data module) and then made a phone call.

$$P_{radio}^{voice} = \begin{cases} P_{scan} & \text{scanning} \\ P_{signal}(bin) & \text{in-service, idle} \\ P_{signal}(bin) + P_{active} & \text{in-service, on} \\ 0 & \text{others} \end{cases}$$

Data Module

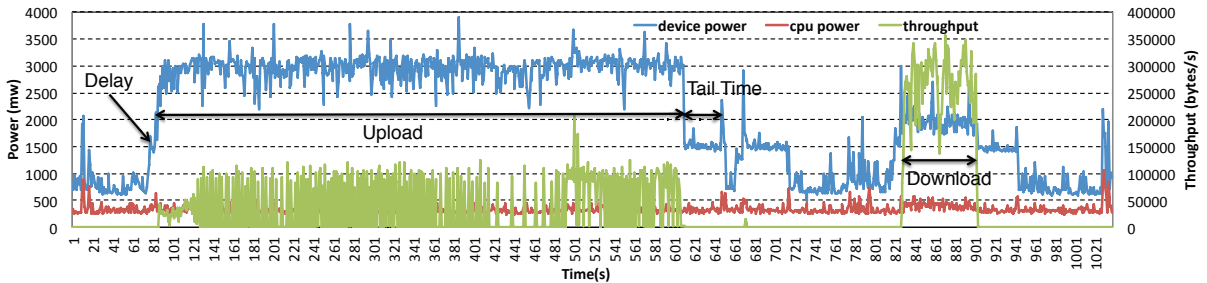


Figure 3.5: The device power and throughput when we uploaded 5 megabytes and downloaded 5 megabytes of data. The carrier is AT&T and network type is HSPA+.

The power consumption of the data module depends on the network type, the carrier, which

defines the tail times, and the data communication state. A common problem with previous work is that they divide the network types into 3G or 4G and then design a power model for each network type. However, the power model of two network types that in the same class may be different. In this section, we only talked about the power model of HSPA evolution, which is technically seen as a 4G network.

Figure 3.5 shows the power consumption and throughput when uploading 5 megabytes of data and then downloading 5 megabytes of data. We notice that different with the power dissipation of LTE network power observed by Huang *et al.* in [48], throughput does not decide the power of the data module. Similar to 3G network type, the power is still related to the RRC state of the data module. The difference is that, when uploading and downloading (in both situations, the data module is in DCH state), the power is significantly different. Moreover, after uploading and downloading, there is a 10-second tail time (in FACH state). Finally, when the data module changes from idle mode to connected mode, there is a delay time of 2 seconds.

Based on this information, we can maintain a state machine to tell the power state of the data module. When the data module has data to send or receive, the state changes from IDLE state to DCH state after 2 seconds delay. Based on the uplink throughput and downlink throughput, we can decide the data module works on high power or low power. If there is no data received or transmitted, the data module changes from DCH to FACH state with no tail time. If the link continues inactive for 10 seconds, the data module changes to IDLE state. Otherwise, if the transmit queue size or receive queue size are higher than a threshold, the data module will be changed to DCH state. However, a common problem with current 3G power models is that they use the bytes received and transmitted in the last interval to decide if the power state should be switched from *FACH* state to *DCH* state because queue size is not available at the application layer.

3.3.3 Display

Display is the most power hungry component in mobile devices, which is inevitably used during user interactions. There are two parameters that influence the power of display: brightness and pixel color. For old LCD technology, brightness determines the power. For the new OLED display, different pixel colors also affect the display power [50]. We evaluated the accuracy and overhead of the two parameters. In the experiment, the brightness level, which is a value range from 0 to 255 (the value of zero means the screen is off), was set from 1 to 255. For each 5-level increase, we measured the power of the system in the idle state. The pixel color is fixed. We observed that the power consumption changed from 336 milliwatts to 1028 milliwatts, and it is linearly correlated to the brightness level. For pixel color aspect, we tested the power of five colors with different brightness levels, as shown in Figure 3.6. For Nexus 4, the power consumption of pixel color follows the order that $G > R > B$. The powers of blue and red are very close. The maximum difference of power consumption is between black and white, and it increases with the brightness level. If the brightness level is under 87 (bright enough for most users), the difference of power is around 60 milliwatts.

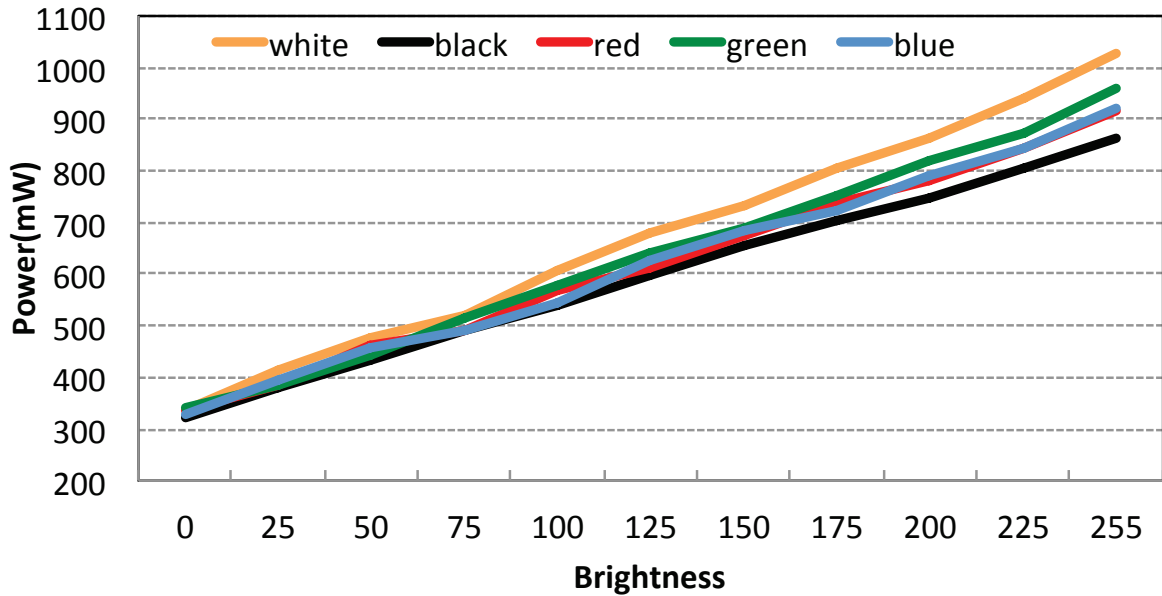


Figure 3.6: The power of different colors with increasing of brightness.

From the overhead point of view, the brightness level is directly read from system file, pixel color information is gathered from a bitmap of the screen shot. We split 1280x768-pixel resolution display into 500 pieces and got the pixel information of central points. It costs 16 ms to calculate display power with pixel information while sampling the whole system state and application-level resource usage without pixel information takes about 50 ms. Then, collecting pixel information will increase the overhead for about 32%, while the maximum gain of accuracy is about 16.45%. Pixel colors are not improper to be used for building OS-layer power profiling service. We consider display power consumption as static power consumption, but not allocate it to the foreground application.

3.3.4 WiFi

Similar to radio, WiFi state decides the power of WiFi. In current mobile systems, WiFi can work in several modes: full mode, high-performance mode, and scan-only mode. In Figure 3.7, we set WiFi to the scan-only mode to generate continuous scan state. In the scan state, the average power of WiFi is about 150 milliwatts. When WiFi is used in full mode and high-performance mode, the time of signal scanning depends on the environment, such as the Internet speed and the number of access points.

The power of WiFi is significantly different when downloading and uploading period. WiFi power is related to the throughput. In the experiment, we gradually increase the upload message size from 5 kilobytes to 1 megabyte. Figure 3.8 shows that the power of WiFi increases from 14.21 milliwatts to 542.84 milliwatts. When there is data transmission, the device power increases while CPU power does not fluctuate too much. In the download case, the trend of CPU power is similar to the trend of packets rate, and the power of WiFi changes from 22.55 milliwatts to 115.82 milliwatts with the same data sizes. The relationship of packets rate and WiFi power in uploading cases is illustrated in Figure 3.9. The power is linearly correlated with packets rate when it is higher than a threshold (e.g. 25 packets/s in the Figure). Since the power difference in uploading and downloading, we distinguish them by comparing the

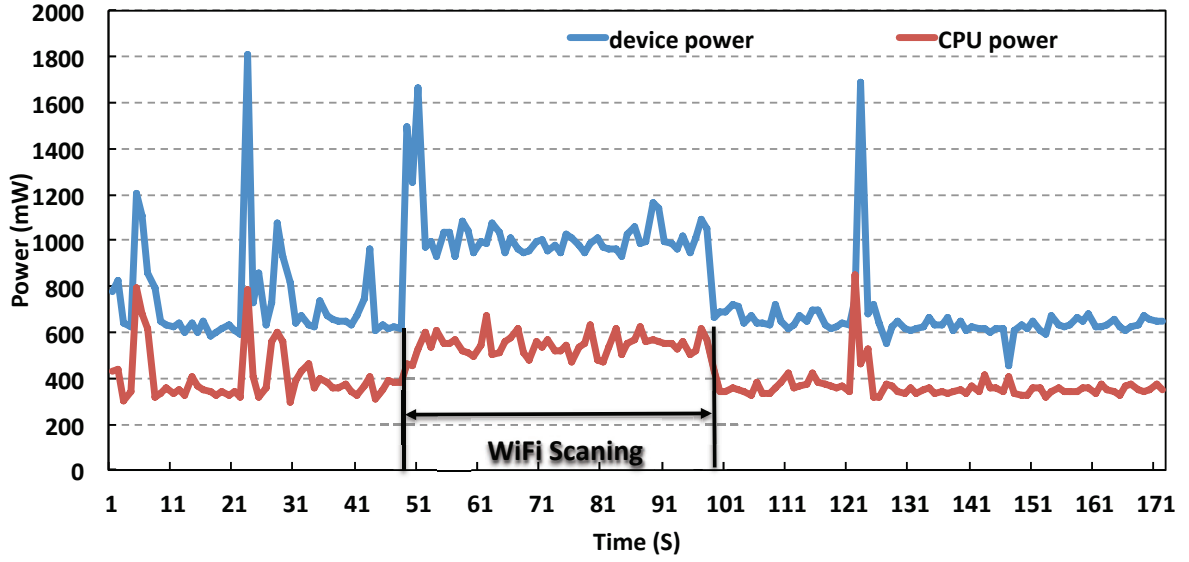


Figure 3.7: The power of the device in WiFi scanning state. WiFi was set to scan only mode.

received packets and the transmitted packets. We allocate the WiFi power to applications and processes based on the percent of data transmitted.

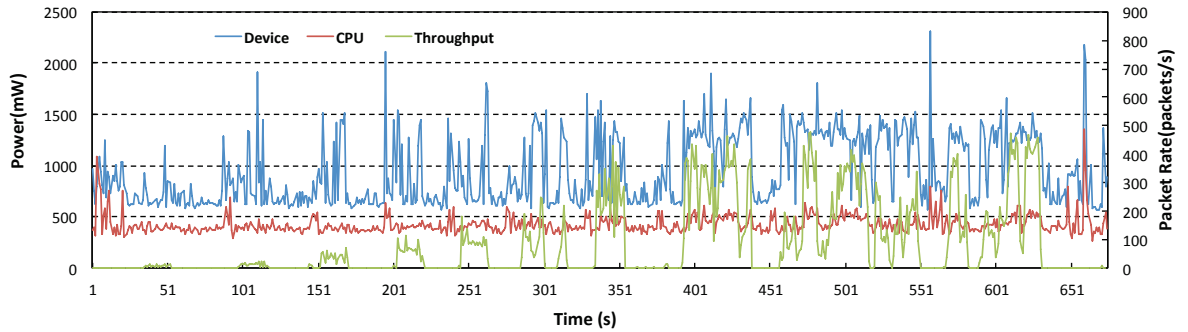


Figure 3.8: The system power and throughput when increasing upload message size from 5KB to 1M.

3.4 Related Work

In previous publications, many power models were proposed to estimate hardware component as well as application power consumption. Most early stage power models were built with hardware performance counters (HPC), which directly related to the state change of hardware components. With a group of HPCs, Isci *et al.* built a power model for process. Bircher *et al.* uses performance counters to build power models for CPU, memory, chipset, I/O, disk, and

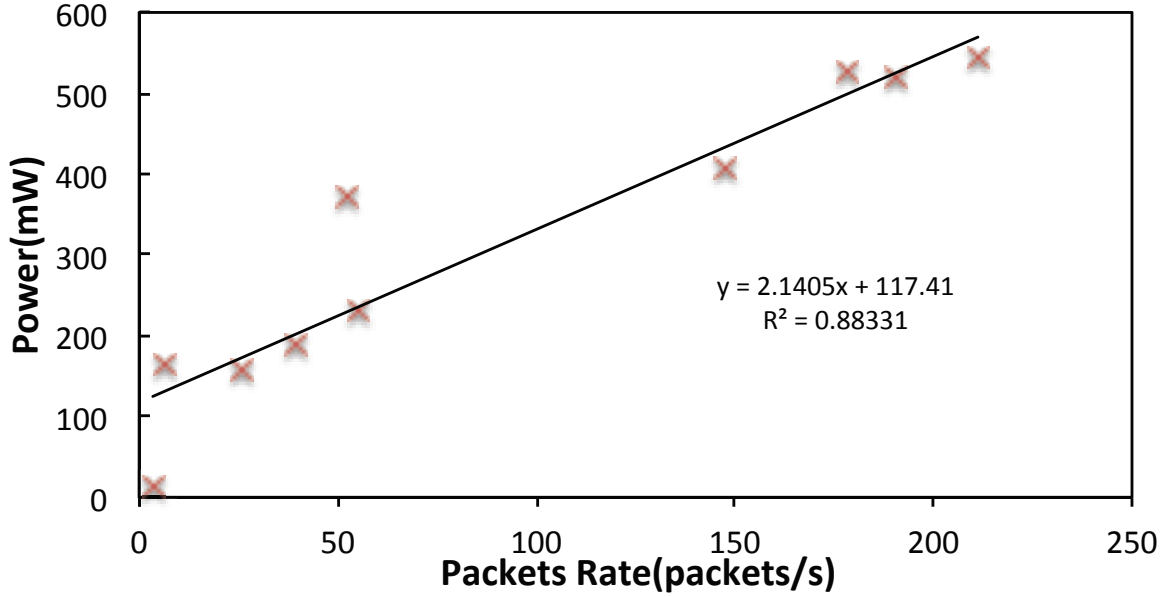


Figure 3.9: The Wi-Fi power and packet rate relationship when uploading data.

GPU [56].

In [52], Zhang *et al.* proposed PowerBooster to automatically construct power models with build-in voltage sensors and charging behavior. This tool builds power models based on hardware power states and resource utilizations. They also developed PowerTutor to estimating application power consumption with the power models found by PowerBooster. In [48], Huang *et al.* built power models for 4G LTE networks based on hardware state. In [50], Dong *et al.* built power models for OLED displays based on screen brightness and pixel colors.

System call, which is the interface between application and kernel, is directly related to hardware states and power. Pathak *et al.* constructed power models based on the traces of system calls [57]. They also introduced *Eprof* to instrument applications' binary code and help developers to analyze the energy consumption per routine in [36]. Similarly, Aggarwal *et al.* as well used traces of system call invocations to estimate hardware power consumptions [79].

Yoon *et al.* monitored kernel activities for hardware component requests to get the usage statistics for the component models [53]. Then, they build power models based on kernel activity. Mittal *et al.* built an energy emulation tool that allows developers to estimate the

energy consumption on their development workstation under various configurations [78]. Ma *et al.* leveraged resource usage statistics patterns and energy information to detect abnormal battery drain issues and suggested corresponding resolutions [80]. Jung *et al.* introduced DevScope [54] to automatically construct power models based on the changes of power state. In [78], Mittal *et al.* built a simulation tool to estimate different hardware components' power as well as application power. The application developer can use this tool to estimate the power consumption of their application on devices of different hardware configuration and environments. Kim *et al.* proposed an event-driven power analysis framework - FEPMA [81]. They collect system events that are related to the power states of hardware components when an application is running on the processor.

3.5 Summary

Application-level power information is essential for evaluating the energy efficiency of energy-saving strategies. In this chapter, we discuss our previous experience about constructing and calibrating power models. We revisited the power models proposed previously and evaluated the effectiveness of the power indicators on our device. The power models were specifically calibrated for Nexus 4 to support our simulation experiment in Chapter 5. In next chapter, we discuss the findings of analyzing 14 users' user behavior. Furthermore, we will talk about how do we analyze the usage pattern of users.

CHAPTER 4

USER BEHAVIOR ANALYSIS

In the early stage of the computer, people used punched cards to tell computers what to do. In the whole process, computers know nothing about the user. Nowadays, however, the way of human-machine interaction significantly changes. Nearly all mobile devices are equipped with various sensors, which are capable of sensing how the user behaves physically or virtually. Moreover, the high computation performance of mobile devices enables systems to analyze and understand users' traits and activities. In our research, we targeted on the interactions between users and applications (user-app interaction). From the monitored user-app interactions, we want to find out the importance of applications in different situations, so that we can dynamically control the state of applications without impacting user experience. In this section, we describe the methods we used to analyze user behavior and discuss our analysis result.

4.1 Introduction

User behavior is a broad concept. As defined in [82], user behavior includes users' attitude, preferences, personality, social relationships, activities, routines and lifestyles. Researchers may concentrate on different aspects of user behavior. For example, Banerjee *et al.* studied the battery usage and charging behavior of users [61]. User interactions, application usage, network traffic and energy drain were researched by Falaki *et al.* through analyzing the usage traces of more than 200 users [64].

In application-level power management, we want to distinguish important applications, so that the power optimization strategies do not influence the execution of these application. Thus, we chose the interactions between users and applications as the research target in this chapter. In [64], the authors classified user activities into two categories: intentional user activity and the impact of user activity. Intentional user activity is directly related to the operations of users. In this chapter, we mainly concentrate on intentional user activities, which represent the subjective consciousness of users. From of this kind of user activities, we can understand

users' preference in applications.

Previous publications [34, 67, 35, 65, 83, 66, 69] proved user behavior is both time-dependent and location-dependent. Thus, we also tried to understand the time distribution and location distribution of user behavior. In the first stage of our research, we monitored the user-app interactions of the first group of users. Based on the sensed user behavior, we found most users frequently used a group of particular application in the specific contexts that marked by time and location. We represent this usage pattern as a group of association rules (or pattern items) between contexts and applications. A pattern item shows how frequently an application was used by the user at the given time and location, and also tells us how confident that similar interactions will happen again. Thus, the related application activities should not be suspended because their results are needed in the following interactions.

4.2 User Behavior Sensing

User behavior sensing is the first step for all the user behavior research. People may use physical or virtual methods to sense user activities. Physical methods use one or several sensors, such as location sensor, gravity sensor, temperature sensor, humidity sensor, the geo-magnetic field sensor, and accelerometer, to sense the required information. However, these sensors are usually power-intensive, and improper for continuous sensing. To probabilize continuous location sensing, we proposed an energy-efficient location provider named *LocalLite* in Chapter 6.3.

4.2.1 Method

We modified Android 4.4.2 to record application usage information. Whenever an application switches to foreground (resume) or switches to background (pause), the system will record the time, *uid*¹ of the application, package name and location to a log file. We built the Android system and flashed the new system image to Nexus 4. In our first stage research, we given the devices to 14 users to use. These users (and their serial numbers) are not totally the same as

¹The unique identifier number assigned to each application in the Android system.

the users that we chose in the second stage data collecting. To distinguish with the users in the other chapters, we label these users as $U1$, $U2$, ..., $U14$ in this chapter. In the second stage, we monitored more data about the system and applications to analyze the power consumption and run the simulation. Chapter 5 describes more details about this.

In the data collecting of both stages, we asked the users to transfer their usage activities to the experiment devices. Before starting to monitor a user's activity, we always reset the system and ask the user to install and setup the applications they used on their personal smartphones. Each user used the device for about a week. When the data collecting of one user finished, we copied the log files from the smartphone. During the experiment, the privacy of users were highly concerned. The serial numbers of users were randomly allocated after collecting all the users' data. The users were also agreed us to use the application usage information in this dissertation.

4.3 Pattern Analysis

In [64], Falaki *et al.* proved the significant difference between the user behavior of users. This observation shows that energy-saving strategies must adapt to the usage pattern of each user, otherwise user experience will be impacted. In this section, we analyze the characteristics of user-app interactions and describe our pattern analysis algorithms.

4.3.1 User-App Interaction

User-device interactions are coarse-grained user activities while user-app interactions enables us to understand application usage. Thus, we use user-app interactions to represent user activities. Our analysis result also shows the significant different between individual user activities. The average number of user-app interactions varies from 21.95 to 223.35 times per day, shown as Figure 4.1.

Even though the great difference between individual user's behavior, we also observed that most users' user-app interactions follow a pattern. For example, Figure 4.2 shows the user-app interactions of user $U10$ were distributed in 19 locations. The user interacted with applications

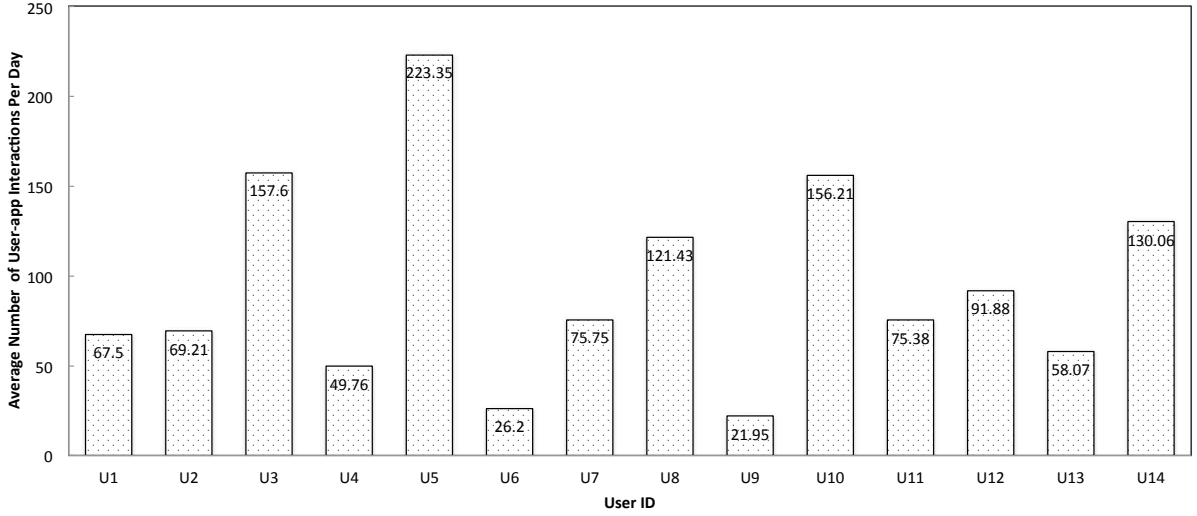


Figure 4.1: The average number of user-app interactions per day of all the users (based the data monitored in the first stage).

periodically and frequently at location 9 and location 17 (The location serial numbers of one user have no relationship with that of the other users). Figure 4.3 shows that user U1, even though generate fewer interactions than user U10, also has a clear usage pattern at location 1. Different to user U10, most of the activities of user U1 were occurred at a single location.

4.3.2 Interest Point

Interest points are locations, such as home and office, that the users use mobile devices frequently. Location can significantly influence user behavior. For example, a user may often use entertainment applications at home while less frequently use them in the office. Thus, location is a fundamental element of usage pattern. The location-dependent characteristic of user behavior was also proved by several previous publications [35, 67, 68, 69, 34]. In this section, we analyze the interest points of users and verify that most users' application usage activities are location-dependent.

To analyze the interest points of users, we developed an algorithm, shown as Listing A.1, with the statistical method. We discover interest points in three steps: (1) count the number user-app interactions occurred at each location; (2) calculate the density of the user-app interactions at each location; (3) calculate the value of core density $density_{core}^{ip}$, which is the threshold

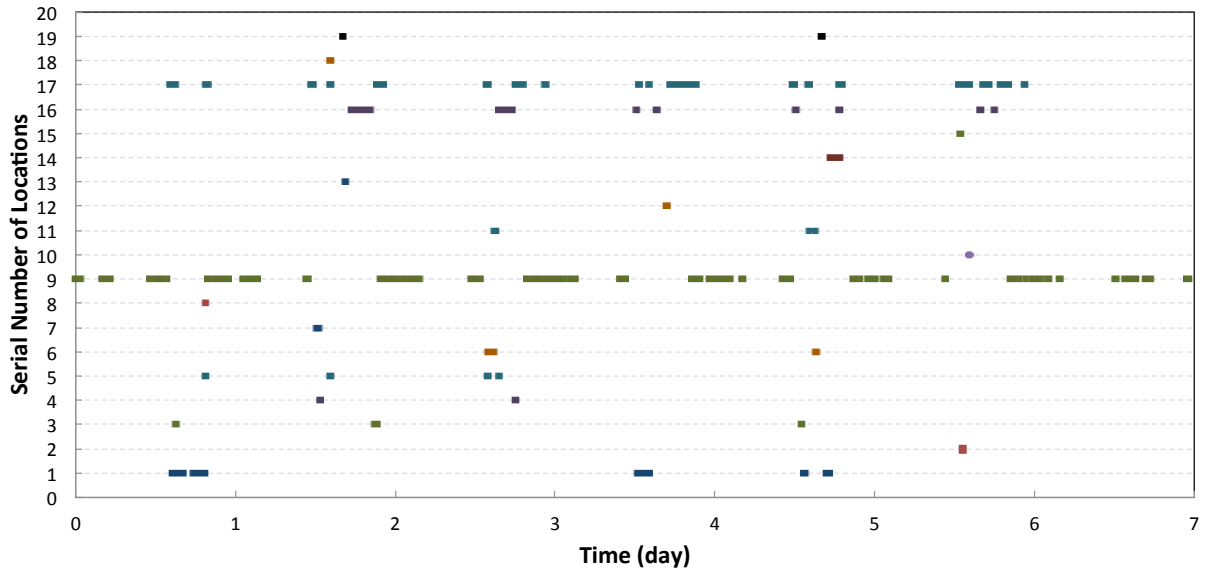


Figure 4.2: The time sequence of the user-app interactions of user U10 at different locations.

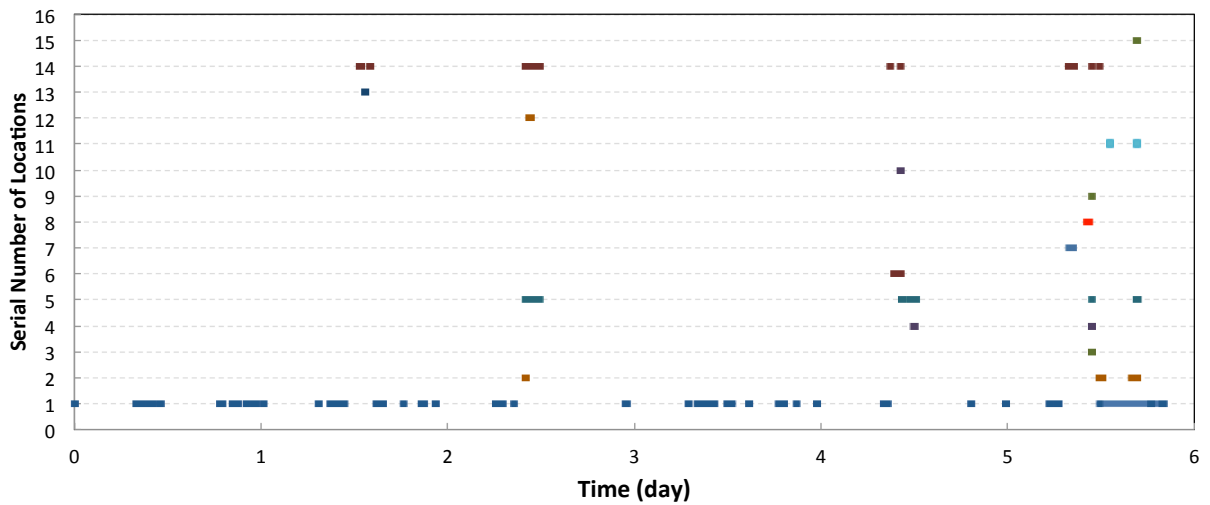


Figure 4.3: The time sequence of user-app interactions user U1 at different locations.

value to identify the location as an interest point, and remove interest points whose density is less than the core density. We use Equation 4.1 to calculate the core density. Equation 4.1a calculates the average density; N_{point} is the number of interest points. In Equation 4.1b, the parameter δ , ranges from 0 to 1, is used to regulate the value of core density. The value of core density ranges from $density_{avg}^{ip}$ to $density_{max}^{ip}$.

The experiment result shows the number of locations varies from 5 to 30, as illustrated in Figure 4.4. Even though some users' activities is distributed in many places, all the users have at least one interest point, shown as Figure 4.4. In terms of location, the result shows that each user's behavior follows a pattern of their own. If the value of δ is smaller, some of the users have more interest points. Also, the system gets more chances to apply energy-saving strategies. However, user experience will be decreased because more low-quality patterns are used. For example, user U13 has two more interest points when δ is 0 than when δ is 0.5. However, from Figure 4.5 we can see that these two interest points only account for about 20.78% of user activities. Furthermore, when δ is 0.5, almost all the users have one or two interest points, and the density of these interest points accounts for more than 32.97%.

$$density_{avg}^{ip} = \frac{\sum_{i=1}^{N_{point}} density_i^{ip}}{N_{point}} \quad (4.1a)$$

$$density_{core}^{ip} = density_{avg}^{ip} + (density_{max}^{ip} - density_{avg}^{ip}) * \delta \quad (4.1b)$$

4.3.3 Active Period

Except usage location, the other aspect of user behavior we concentrated on is the active period of users. In an active period, the user interacts with applications much more frequently than in other periods. From Figure 4.2 and Figure 4.3 we can see that both users have several dense user-app interactions groups. Besides, we notice that the occurrence of the user-app interaction group is periodical at some locations. This observation proves that usage location

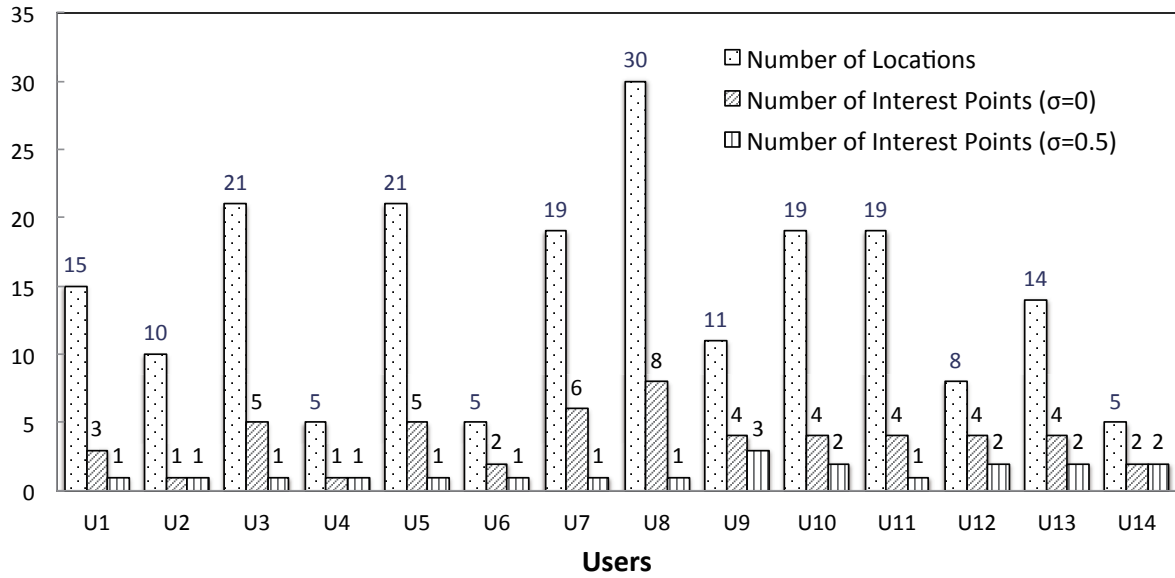


Figure 4.4: The number of locations and interest points of the users (based the data monitored in the first stage).

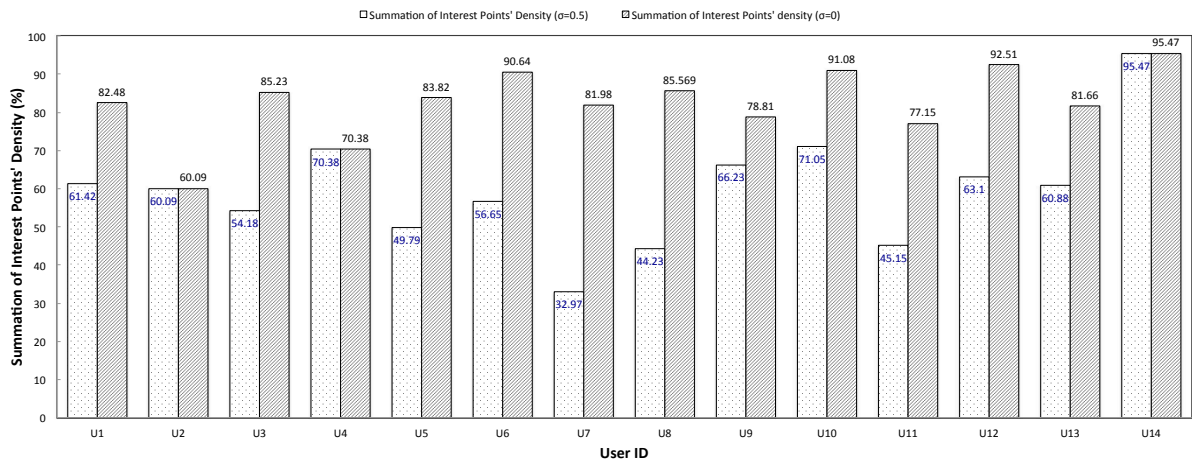


Figure 4.5: The summation of the density of the interest points (based the data monitored in the first stage).

and active period of users are interrelated.

We used the DBSCAN (density-based spatial clustering of applications with noise) [84] algorithm to find the active periods. The DBSCAN algorithm is a density-based clustering algorithm that groups together tightly packed points (in our case, a point is a time slice in the sensing time unit). The sensing time unit can be one day or one week. A few sensing time units comprise a sensing period. We equally split the time unit into multiple 10-minute time slices. If a user-app interaction occurs in a time slice, we add 1 to the interaction number of this time slice. If a user-app interaction crosses several time slices, we add 1 to the interaction number of all the crossed time slices.

Listing A.2 shows the implementation of this algorithm. It analyzes active periods in several steps:

1. Equally split the time unit (a day or a week) into 10-minute time slices.
2. Iterate each user-app interaction, and count the number of user-app interactions in each time slice (or point). If an user-app interaction spans several time slices, the interaction number of each time slice increases 1.
3. Calculate the density of time slices and mark the time slices as core point, border point or noise point based on two threshold values $density_{border}^{tp}$ and $density_{core}^{tp}$. We use the Equations 4.2 to calculate these two thresholds.
4. Iterate all time slices. If a time slice is a core point, we allocate it to a new cluster. If a time slice is a border point, we find the nearest core point within 1 hour and assign this point to the core point's cluster.
5. Iterator all points again. Group all consecutive points that belong to a cluster into a new cluster. If the start time of the current new cluster is 30 minutes larger than the end time of the previous cluster, we merge them together.

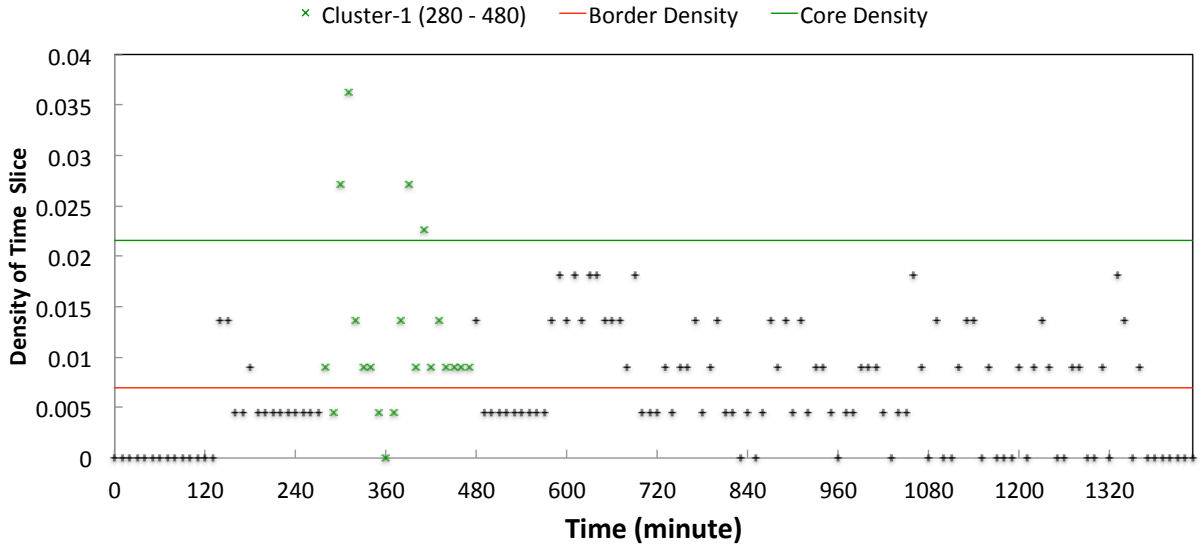


Figure 4.6: The density of the time slices of user U3 and the corresponding clusters.

Different with the regular implementation of DBSCAN algorithm, we merge consecutive clusters that are within 30 minutes to decrease the number of small clusters. This design not only decreased the number of usage patterns but also helpful to user-centric system design because user experience gets more chances to be considered (when applying mechanisms).

Figure 4.6 and Figure 4.7 show the density of each time slice of user U3 and user U8. In Figure 4.6, four time slices are labeled as core points, and all of them are categorized to cluster 1. The user-app interactions are scattered very evenly in the other time slices, and thus none of the time slices is labeled as core points. Because there are no core points, the algorithm did not find any active periods except cluster 1. Different to user U3, the core points of user U8 are distributed in three different periods, shown as Figure 4.7. According, the algorithm found three active periods for user U8. In these two figures, we found some of the noise points are counted into the clusters, which is caused by the last step of the algorithm. In this way, we can effectively decrease short active periods.

Figure 4.8 shows that each user has at least one active period. The active period of some users, such as user U6 and user U9, is very short. That is either because they are not using the device frequently, or because their user-app interactions distributed very evenly. We notice that

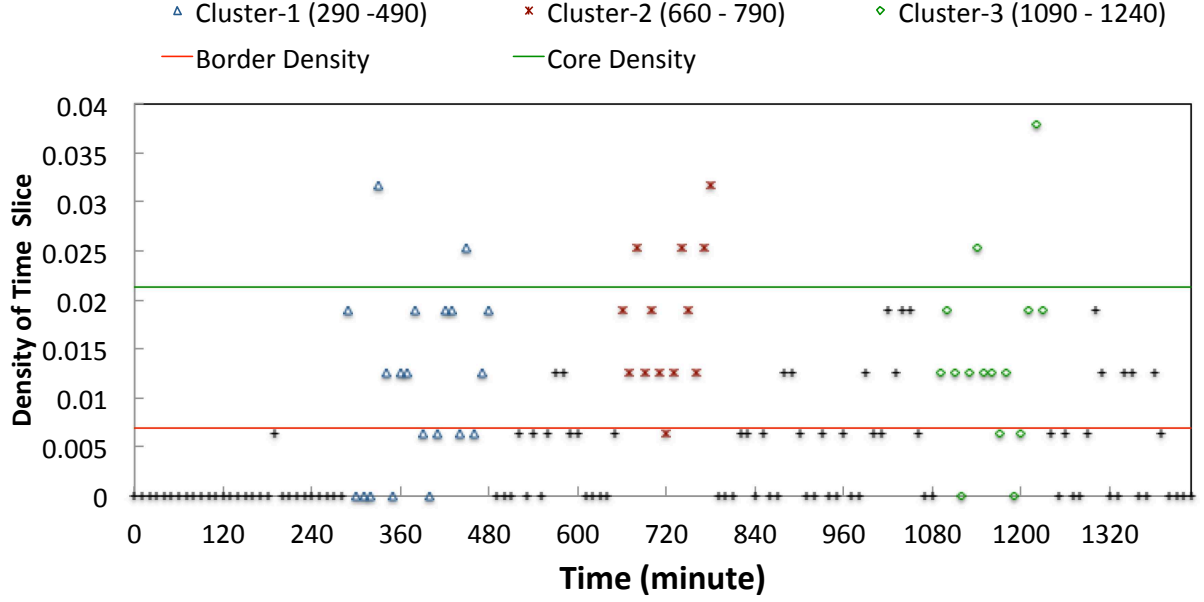


Figure 4.7: The density of the time slices of user U8 and the corresponding clusters.

nearly all the users' active periods are different with each other. This observation shows us that the system should be designed based on user behavior. Only in this way, we can satisfy both user experience and battery life.

$$density_{border} = \frac{1}{N_{ts}} \alpha \quad (4.2a)$$

$$density_{core} = density_{border} + (density_{max} - density_{border}) * \beta \quad (4.2b)$$

4.3.4 Usage Pattern

In the previous two sections, we described the method of analyzing interest points and active periods of users. The usage pattern of a user is composed by a group of pattern items, each of which includes an interest point, an active period and the application. Except that, we need to evaluate the quality of the pattern item. We use Equation 4.3 to calculate the support (the usage frequency of the application) of the pattern item. The support shows how frequently the user interacts with an application at the given location and active period. However, the

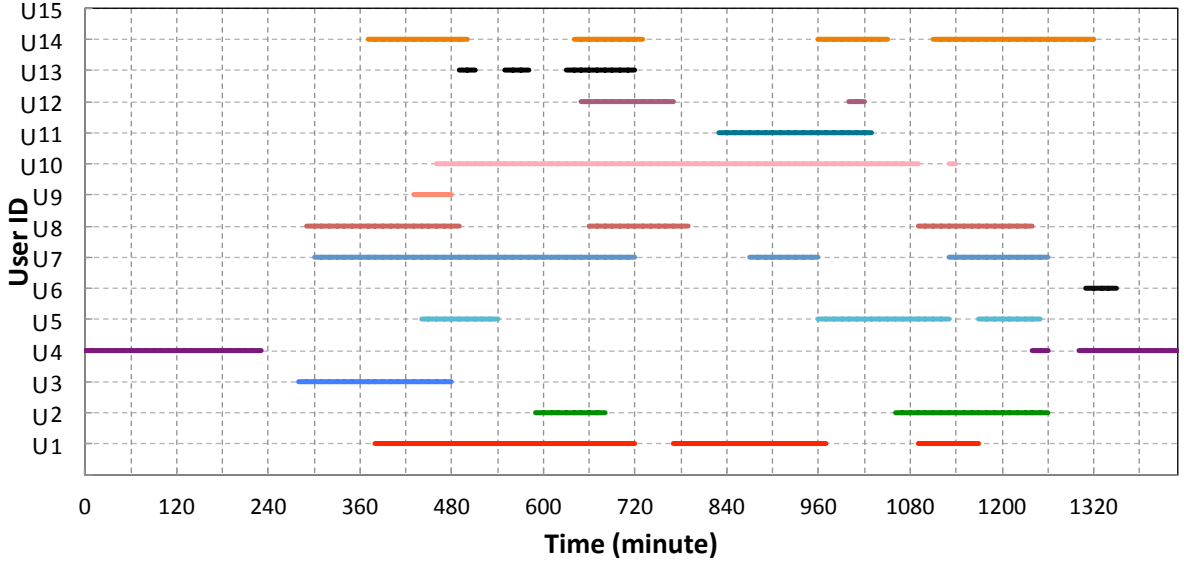


Figure 4.8: The active periods of all the users (based the data monitored in the first stage).

usage pattern of some users may change a lot with time. In the real usage case, the time window of data used to analyze the usage pattern should not last too long. Otherwise, the usage pattern may fail to distinguish which application is important.

$$support^{pattern} = days_{occured} / days_{total} \quad (4.3)$$

After we know interest points, active periods, and applications, we find out each combination of these three elements and count the number of user-app interactions of each combination. The computation is not CPU-intensive because the number of interest points and active periods are limited. Finally, we calculate the support of each pattern. We use a threshold $TH_p (0 < TH_p < 1)$ to filter unqualified usage pattern. Listing A.3 shows the algorithm to analyze the usage pattern. Table 4.1 displays the usage pattern of user U1. From this table, we can see that this user uses more applications frequently in the morning. In the afternoon and evening, the user uses fewer applications often.

Table 4.1: The usage pattern of user U1.

Context \ Apps	6:20AM - 12:00PM	12:50PM - 16:10PM	18:10PM - 19:30PM
	Loc 1		Loc 2
Seetings	33.3%		
Launcher	100%	50%	66.7%
Google Play Store	33.3%		
QQ	50%		
Chrome	66.6%		
Facebook	66.6%		
Weather	33.3%		
USA Today	66.6%		
Weishi	100%		33.3%
Youtube			83.3%

4.4 Related Work

Mobile devices, such as smartphones and wearable devices, have become an indispensable accessory in our daily life. Our mobile devices know the places we visited, applications used, network/cellular data used, phone calls made and so on. Thus, we can collect a considerable amount of user behavior metadata with them. Many previous publications [64, 65, 66, 35, 67, 68, 69, 34] used this metadata to analyze the pattern of individual user's behavior or user groups' behavior. Most patterns are either represented as the occurrence frequency of an event or the association rules of several events.

In [64], Falaki *et al.* collected the traces of 255 users to study their smartphone usage behavior. This research was mainly concentrated on internal user-device interactions, user-app interactions, and the impact of these interactions on energy usage and network usage. They found the vast diversity between the individual user among nearly all aspects of research. For example, the cellular data usage per day varies from 1 MB to 1000 MB. Depending on their experiment result, they proposed to improve user experience and energy consumption from adapting to user behaviors. Rahmati *et al.* collected the usage activities of 14 teenage users [65]. Their research targeted at smartphone usage, application usage, and usage behavior evolution. They found these users' usage is highly mobile and location-dependent, and the

usage pattern changes with location. Moreover, the experiment shows the social purpose is the main driving force of the smartphone usage, and most users' usage behavior converges after they get familiar with the functions of the devices and applications.

Srinivasan *et al.* proposed a system service named MobileMiner [66] for tracking user behavior and analyzing usage patterns. They developed a weighted mining algorithm - WeMiT to examine the association rules of user behavior related events. With this algorithm, they analyzed the user behavior data collected from 106 users and observed several interesting patterns, such as the outgoing phone call pattern and cellular network usage pattern. They describe the usage scenarios of several patterns to improve user experience. For example, the outgoing phone call pattern can be used to predict the next user to call, which is helpful for the UI design of the contact application. In [35], Shin *et al.* analyzed the correlation between application usage and its related context. They tried to predict the probability that the user uses an application in the current context. So that, they can organize the order of application icons on the home screen and improve user experience. In [67], Parate *et al.* proposed the APPM (App Prediction by Partial Match) model to predict the next application the user will be most likely to use. Further, they introduced the TTU (Time Till Usage) model to estimate the appropriate time to prefetch content. Different with other works, their method does not require any power-hungry contextual data.

Nath *et al.* developed ACE [68] to sense user context. They used the speculative sensing technique to learn the relationship between context attributes, such that low energy requirement context attributes can be used to speculate high energy requirement context attributes. Böhmer *et al.* made a large-scale deployment-based research with more than 4,100 Android users [69]. They logged the detailed application usage information of these users, and found several interesting group usage patterns, such as News applications are popular in the morning and games are popular at night. In [34], Do *et al.* researched two key contextual cues, location and proximity, of users, and found several import patterns that are helpful for understanding

how users use their smart phones.

Most of these previous works tried to understand how users interact with mobile devices and the evolution of their usage pattern. Even though some publications [64, 65, 66, 35] propose to utilize user behavior to improve user experience (e.g. help the user to find the next application to use quickly), not too much work are done to improve the battery life of mobile devices. Different with previous user behavior related research, we use the usage pattern to make the energy-saving decision.

4.5 Summary

The user behavior we are interested is how the user interacts with applications. The location and time of the user-app interaction are two of the key characters of user behavior. In this section, we use statistical and data mining techniques to analyze the interest points, active periods and usage pattern of users. The analysis result shows that even though individual user's user behavior may be significantly different with each other, most users' behavior follows their pattern.

From the usage pattern of users, we can distinguish redundant and unimportant application activities. In the next chapter, we propose a user-centric power management and an energy-saving strategy that used the task grouping technique to optimize the power consumption of background applications.

CHAPTER 5

USER-CENTRIC POWER MANAGEMENT SYSTEM

In the previous chapters, we analyzed the deficiencies of current power management systems and discussed the important observations that motivated us to design the user-centric power management system. In this chapter, we describe the design and implementation of the *UPS* system and the *UCASS* strategy in detail.

5.1 Overview

The power management systems of most traditional operating system were implemented based on the ACPI (Advanced Configuration and Power Interface) specification [33]. Device components can work in different performance states, each of which requires a different power consumption. The power managers of the operating system dynamically change the states of device components based on the performance requirement of tasks. Usually, the power managers use performance metrics, such as CPU utilization, to represent the performance requirement of tasks. For example, the *CPUIidle* mechanism [15] decides the sleep state (C state) of the processor based on how long the CPU had worked in the idle state (CPU utilization is zero), and the *CPUFreq* mechanism [12] dynamically set the frequency and voltage of the the processor based on the workload of the CPU.

Ideally, the system uses as much power as the tasks need. However, it is nearly impossible to predict the performance requirement accurately; power managers normally set the new power states of device components based on the instantaneous values of one or several performance metrics. These values usually cannot comprehensively represent the performance requirement of device components. Modern mobile operating systems, such as Android, have the same problem because most of them are running on top of the Linux kernel. Moreover, application and system activities cause these power managers cannot work effectively. Our previous analysis found that a considerable amount of background activities were not generated by the proactive user operations or even not subjectively required by the user. These activ-

ities wasted a large amount of battery energy. Existing power management systems, however, cannot eliminate the energy loss caused by these redundant activities.

To solve the power problem of mobile operating systems, we need to manage power consumption at application-level. We name this kind of method as application-level power management, which is a necessary supplementary of traditional power management systems. Application-level power management uses power optimization techniques, such as energy adaptation, task grouping and computation offloading, to dynamically change the state of applications and the system. For example, the latest iOS operating system uses the energy adaptation technique to extend battery life. When the battery level is lower than 20%, the system can work in low-power mode, which alters system behavior, such as visual effects, and application behavior, such as background application refresh, to save battery energy. However, this kind of uniform energy-saving strategies cannot satisfy all the users. It may significantly impact user experience. How to balance user experience and energy-saving is the main issue for designing energy-saving strategies.

In this dissertation, we solved this problem with the usage pattern of users. From the analysis result of user behavior in Chapter 4, we found the behavior of the individual user was significantly different, and but the user behavior of most user are time-dependent and location-dependent. All the users have a usage pattern of their own. Other publications [64, 65, 66, 35] also observed these phenomenons. We can take advantage of the characteristics of user behavior to dynamically customize the energy-saving strategies for users. The strategies that adapted to the usage pattern of each user can effectively balance user experience and battery life. We name this kind of power management as user-centric power management. Similar to the procedure of pervasive computing [85], user-centric power management also tracks, analyzes and utilizes user behavior. The difference is that we use it to eliminate the wasted energy and supply a better battery life to users.

We designed and implemented our idea in the *UPS* system. It monitors user behavior

and analyzes the usage pattern of users. As a user-centric power management framework, it bridges user behavior with energy-saving strategies. It notifies these strategies whenever the system state (battery state, charging state, screen state), active period or usage pattern changes. Such that, researchers can concentrate on designing power-optimization policies. We also designed the *UCASS* strategy to handle the redundant application activities, and integrated it into the *UPS* system. This strategy eliminates the wasted battery energy caused by background applications through task grouping and rescheduling. In this chapter, we elaborate the design, implementation and evaluation of the *UPS* system as well as the *UCASS* strategy.

5.2 Background

We propose the *UPS* system and the *UCASS* energy-saving strategy based on several key observations about user behavior, mobile devices, and mobile applications. In this section, we describe these observations and discuss why these observations are important to optimize the battery drain of mobile devices.

5.2.1 Characteristics of User Behavior

The first observation of user behavior is that user-app interactions are time-dependent. From the analysis in Section 4.3.3 of Chapter 4, we found users used the device and interacted with applications much more frequently during some active periods. This observation is an important hint for designing the system energy-efficiently. We can setup the energy-saving strategies to behave differently when the current time is inside or outside an active period. For example, we can dynamically set the display timeout time to a larger value during the active periods, and set it to smaller value during other periods. In this way, we can save the energy consumption of display without influence user experience too much.

Moreover, we found users frequently generated user-app interactions in several dedicated locations (interest points), most which are WiFi available. When the user uses his or her device at the same location, the information acquired from some system services, such as location service, is consistent or inter-related. Thus, we can optimize the power consumption of these

services through eliminating the duplicated computations. The design of the *LocalLite* location provider, described in Chapter 6, follows this observation.

The users were also inclined to use a particular group of applications in some contexts that are marked by active period and location. This usage pattern can form a group of association rules in the form of Equation 5.1. The support of these rules shows us how frequently the user used the application or how significant the application is to the user in the context. Based on this phenomenon, we can distinguish important applications and unimportant applications and treat them differently, such that energy-saving strategies do not impact user experience significantly. Furthermore, it is helpful for user-centric system design, such as improving the UI of the system [35]. So that, the user can quickly find out the application that he or she will use next.

$$(location, active\ period) \stackrel{support}{\Rightarrow} application \quad (5.1)$$

5.2.2 Characteristics of Mobile Devices and Applications

The availability of plenteous applications, the majority of which are developed by third-party developers, is one of the most conspicuous characteristics of modern mobile devices. Compared with old-styled mobile devices, users usually installed more applications on the new-generation devices. Besides, nearly all the mobile devices are integrated with multi-core processors and support real multi-task processing. Normally, one application runs in the foreground and occupies the whole screen, while most other applications run in the background. The tasks of the foreground application are the most important tasks to the user since the user usually expects to see the result immediately. For example, a web page loading task is expected to display the web pages on the screen as soon as possible. This kind of user-app interactions, in which the result of the task blocks the user to take other operations, is synchronous.

On the contrary, the interactions of all the background applications are asynchronous. Usually, the user will be notified to check the result when a background task finishes or the user

gets the result during a synchronous user-app interactions in the future. However, in most of the other cases, the results of the background tasks are even not noticed by the user. Only a small part of background tasks are expected by the user while most others are a waste of energy. This situation can be improved through either suspending the execution of these tasks or rescheduling these tasks to the next device charging or active time.

The other significant characteristic of modern mobile devices is the new-generation high-performance communication modules (3G/4G or LTE). These hardware components supply much faster network speed than old-styled devices. Besides, wireless networks are more commonly deployed. Connecting to the network through various wireless networks is much more convenient and economic than ten years ago. These new techniques promote the development of mobile computing. Nowadays, majority mobile application, such as social networking applications, video players, and email clients, have to retrieve data from and upload data to cloud services. However, these wireless communication modules are usually very power-hungry, and network communication consumes a large amount of battery energy. Thus, many previous work try to optimize the power consumption of communication components.

Normally, these communication components are designed with several power states, and the components drain much more power when the device is active. The communication components usually have a tail time (T_{tail}) after processing all the data transmission tasks. The hardware component is switched to a lower power state if the inactive time is longer than the tail time. On the contrary, whenever a new network activity occurs during the tail time, the component switches back to active mode immediately. The power consumption in the tail period is also extremely high such that the state-switching time is very short. Task grouping can effectively decrease the tail time power consumption of the device.

5.3 System Design

Based on the previous analysis, many aspects of the mobile system can be optimized for energy-efficiency. A user-centric power management system is strongly needed to save battery

energy while at the same time decrease the impact on user experience. In this section, we propose a **User-centric Power management System (UPS)** for mobile operating systems. The *UPS* system is an open power management framework that bridges the space between user behavior and energy-saving strategies.

5.3.1 Architecture

From the point of data processing, we can divide the task of the *UPS* system into three subtasks. First, it monitors user-app interactions, each of which includes the corresponding time, location and application information, and saves this information into a dataset. Then, it uses statistical and data mining algorithms, that we discussed in Chapter 4, to find out the usage pattern of the user from the dataset. Finally, the usage pattern is applied in energy-saving strategies to make power optimization decisions. Since the strategies consider the user behavior of individual user, they can effectively trade off energy-saving against user experience. Figure 5.2 shows the time windows that perform these three subtasks. We will discuss the relationship of them in the following sections.

As illustrated in Figure 5.1, the *UPS* system is composed of four main components: user behavior monitor, usage pattern analyzer, *UPS* power manager and energy-saving strategies. The user behavior monitor listens to user-app interaction events, and stores the information of the event into database. The usage pattern analyzer uses the monitored user behavior as input and analyzes the usage pattern of the user. The *UPS* power manager works as the central controller of the system. It not only schedules the pattern analysis task, but also listens to several system events and triggers the corresponding methods of energy-saving strategies. Whenever an energy-saving strategy receives a system event, it makes energy-saving decisions based on the usage pattern.

5.3.2 User Behavior Monitor

Monitoring user activities is the first task of designing all kinds of user-centric systems. As shown of Figure 5.1, the user behavior monitor listens to user behavior events that are

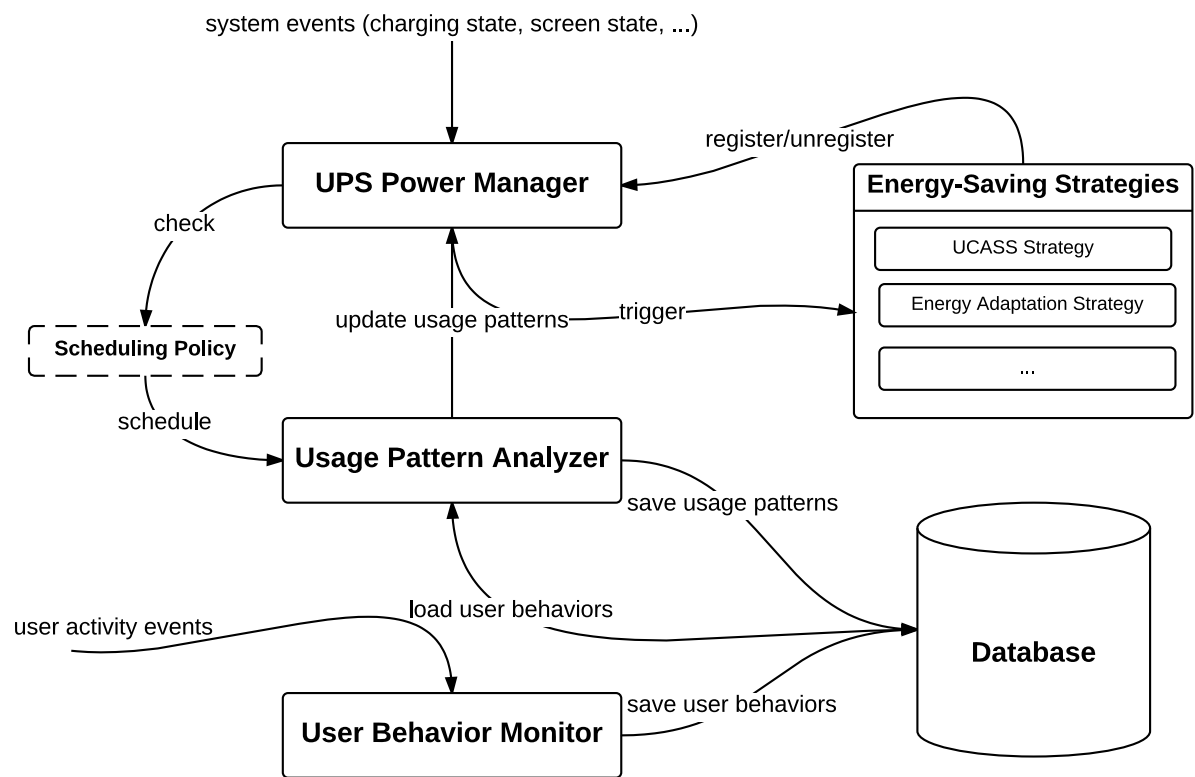


Figure 5.1: The architecture of the UPS system.

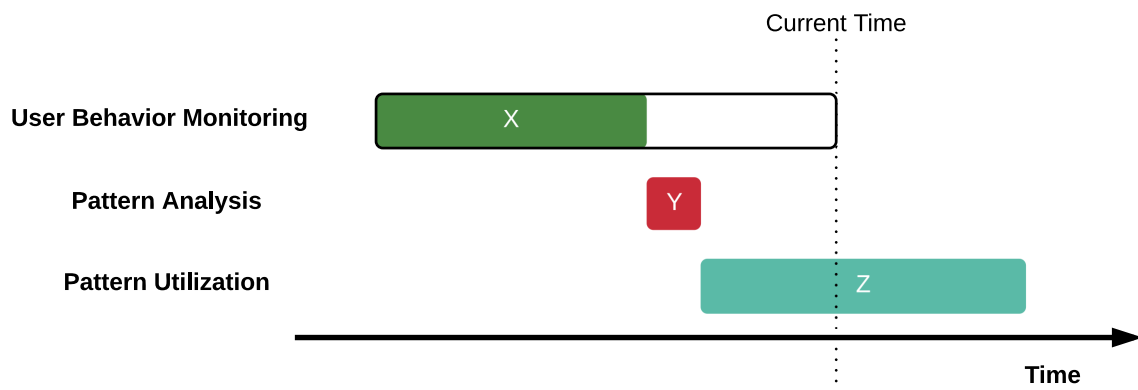


Figure 5.2: The time windows of the three steps in the data processing procedure of the UPS system.

emitted by the system and then saves the corresponding information to the database. In the *UPS* system, user behavior is represented as user-app interaction. More specifically, we care about when and where the user opens an application and closes an application because the time distribution and location distribution are two significant characteristics of user behavior. Previous publications [65, 66, 35, 69] also proved user behavior is time-dependent and location-dependent.

A user-app interaction event contains the information such as the event type (*RESUME* or *PAUSE*), time information, location information, and application information. Existing mobile operating systems do not supply this event; we implemented it in Android for the *UPS* system. We will discuss the implementation in Section 5.4. The *RESUME* event will be received when an application is opened (a user-application interaction starts). On the contrary, the *PAUSE* event occurs when the application is closed or moved to the background (the user-app interaction finishes). A *RESUME* event and the followed *PAUSE* event of the same application define a user-app interaction.

Because the user behavior monitor collects the location of user-app interactions, we designed a light-weight location provider to optimize the power consumption of location service. We will discuss the design of this location provider in Chapter 6. Besides, all the events received by the user behavior monitor are asynchronous, and the frequency of the events is very limited compared to other system events. Such that, user behavior monitor does not cause too much energy consumption in the system.

We only keep the most recent data, illustrated in Figure 5.2, because the usage pattern of users may change with time. Outdated user activities are much less relevant to the current usage pattern of users than new user activities. The user behavior monitor deletes these outdated records periodically. Besides, we can safely assume the usage pattern does not change in a short time. In the time window **Y**, the usage pattern analyzer uses the data recorded in the time window **X** to get the current usage pattern. In the existing design, we set the length of time

window X as a month. In the future, longer user behaviors should be monitored and analyzed to find the better time window for each user.

5.3.3 Usage Pattern Analyzer

In Chapter 4, we designed several statistical and data mining algorithms to analyze usage pattern. The usage pattern analyzer implements these algorithms, and periodically executes the analysis task. It uses the user-app interactions occurred in the time window X as inputs, and generates a group of pattern items. Each pattern item represents how frequently the user used an application in the specified period and at the specified location; it also denotes the confidence that similar user-app interactions will occur again. We can formally define it as an association rule, as shown of Equation 5.1. After the new pattern is generated, it notifies the *UPS* power manager to update its state with the new usage pattern, which will be used in the following time window Z . Moreover, other system services can also listen to the update event of usage pattern and utilize it in user-centric system design. The user behavior analyzer saves the result in the database, such that it can load the usage pattern and notify the listeners when the system starts.

The *UPS* power manager schedules the analysis task based on the scheduling policy we defined. From the previous analysis of user behavior, we found most users charge their devices at least once a day. The scheduling policy only schedules the analysis task while the device is charging. Scheduling the task when charging only slightly extends the charging time; it rarely has any influence on the user experience. When we design energy-saving strategies, we also utilize this characteristic of mobile devices. Moreover, we assume the usage pattern will not change significantly during the time interval T_s (default value is 24 hours), which is the average pattern analysis interval. Overall, the *UPS* power manager schedules the analysis task whenever the following two rules are satisfied.

1. The device is connected to the power supply.
2. The time interval between the current time and last scheduling time is larger than T_s .

5.3.4 UPS Power Manager

The *UPS* power manager is the central unit of the *UPS* system. It listens to various events and triggers energy-saving strategies to update state. For example, when the device connects to the power supply, an energy-saving strategy may be turned off. These events include not only system events, such as the charging state event and the screen state event, but also internal events of the *UPS* system, such as the update event of usage pattern and active period. The *UPS* manager supplies a group of user-level APIs for users to set up the *UPS* system and to retrieve the status information of the *UPS* system. For example, users can enable or disable an energy-saving policy through the *setStrategyStatus* API, and the *getStrategies* API returns the available energy-saving strategies.

void setPowerManagerStatus(boolean status)

Set the status (On or Off) of the UPS power manager service.

boolean getPowerManagerStatus()

Retrieve the status (On or Off) of the UPS power manager service.

ArrayList getStrategies()

Return the list of available energy-saving strategies.

void setStrategyStatus(String component, boolean status)

Set the status (On or Off) of an energy-saving strategy specified by the “*component*” variable.

boolean getStrategyStatus(String target)

Get the current status of an energy-saving strategy specified by the “*component*” variable.

void setParameters(in ParameterSet paramSet)

Set the parameters of a component of the *UPS* system, which is specified by the “*component*” field of the *ParameterSet* object. The input “*paramSet*” includes a group of

Parameter objects, each of which defines a system parameter of the specified component.

ParameterSet getParameters(String component)

Retrieve the parameters of a component specified by the “*component*” variable.

void setParameter(String component, in Parameter param)

This method sets a parameter of the specified target. The *Parameter* class includes a key-value pair, and the key specifies the parameter to set.

Parameter getParameter(String component, String key)

This method returns the a parameter of the specified component and key.

long getLastScheduleTime()

Return the last scheduling time of the user behavior analyzer.

The UPS power management system is triggered by system events. When the power manager receives an event, the state energy-saving strategies may also have to be updated. To hook up energy-saving strategies and the power manager, we defined the following two APIs for the strategies.

boolean listensToEvent(Event event)

Returns true if the strategy needs to update itself when this event is received. Otherwise, returns false.

void update(Event event)

Energy-saving strategy implements the power-aware policy in this method. It makes different operations based on the new event received.

When the power manager receives a new event, it iterates the registered energy-saving strategies. First, it invokes the *listensToEvent* method to check whether the strategy listens to

the event. If a strategy listens to this event, the power manager invokes the *update* method of the strategy to trigger the energy-aware operations.

5.3.5 User-Centric Application Sieving and Scheduling

In the UPS system, we designed a default energy-saving strategy - UCASS (user-centric application sieving and scheduling), which saves energy through task grouping and rescheduling. Different to the previously mentioned task grouping methods [44, 45, 51], it utilizes the usage pattern of users to decide how to reschedule the tasks. We designed the application sieving rules to check applications and handle the tasks of these applications energy-efficiently. In the different situation, we may choose to schedule the tasks of an application at a different time, such as device charging period or device active period.

The UCASS strategy groups and reschedules tasks based on user behavior. In Chapter 4, we use statistical, and data mining techniques to analyze the usage pattern of the user. The usage pattern tells us how frequently each application was used at a specific location and time. Since the usage pattern of most users does not change frequently. Thus, the usage pattern implies postponing which applications' background tasks cause less impact on user experience. Based on this hint, the UCASS strategy categorizes background tasks into several groups and decides when to reschedule the tasks in each group.

Task Scheduling

As described in Section 4.3, we use the *support* of the pattern item to represent the confidence that similar interactions will happen again. The support of a pattern item shows how frequently an application was used in a context. An application may have multiple pattern items, each of which has a different context. If the support of a pattern item is larger than the threshold TH_p , we mark this pattern item as valid. In this section, the usage pattern we talked about are all valid because the user behavior analyzer filters invalid pattern items with TH_p . Based on the support of pattern item and a threshold values TH_s , we defined three scheduling

strategies:

1. Scheduling the task immediately.
2. Scheduling the task when the device is active. Add the task to the AE queue.
3. Scheduling the task when the device is charging. Add the task to the CE queue.

Figure 5.3 illustrates the flowchart of making scheduling decisions. From this figure, we can see that the *UCASS* strategy also considers device status and application status. When the device is charging, all tasks will be scheduled immediately. When the device is in charging state, the power consumption of the tasks can be ignored. Then, it checks whether the screen is on (device active state). When the device is active, the tasks of the foreground application will be execute immediately. The tasks of background applications will either be scheduled when the device is charging or scheduled immediately based on whether a pattern item exists for the application in the current context. If the screen is turned off (device idle state), it checks whether the pattern item exists and whether the support is smaller than TH_s . If the pattern item does not exist, it schedules the task to execute in the charging state. If the support is smaller than TH_s , it schedules the task to execute when the device is active. On the contrary, the task is executed immediately.

Trigger Events

The following listed events trigger *UCASS*.

- *SCREEN_ON*: The device becomes active. It schedules the tasks in the AE queue to execute.
- *SCREEN_OFF*: The device becomes idle. If the AE queue is not empty, it stops to schedule the remaining tasks in the AE queue.

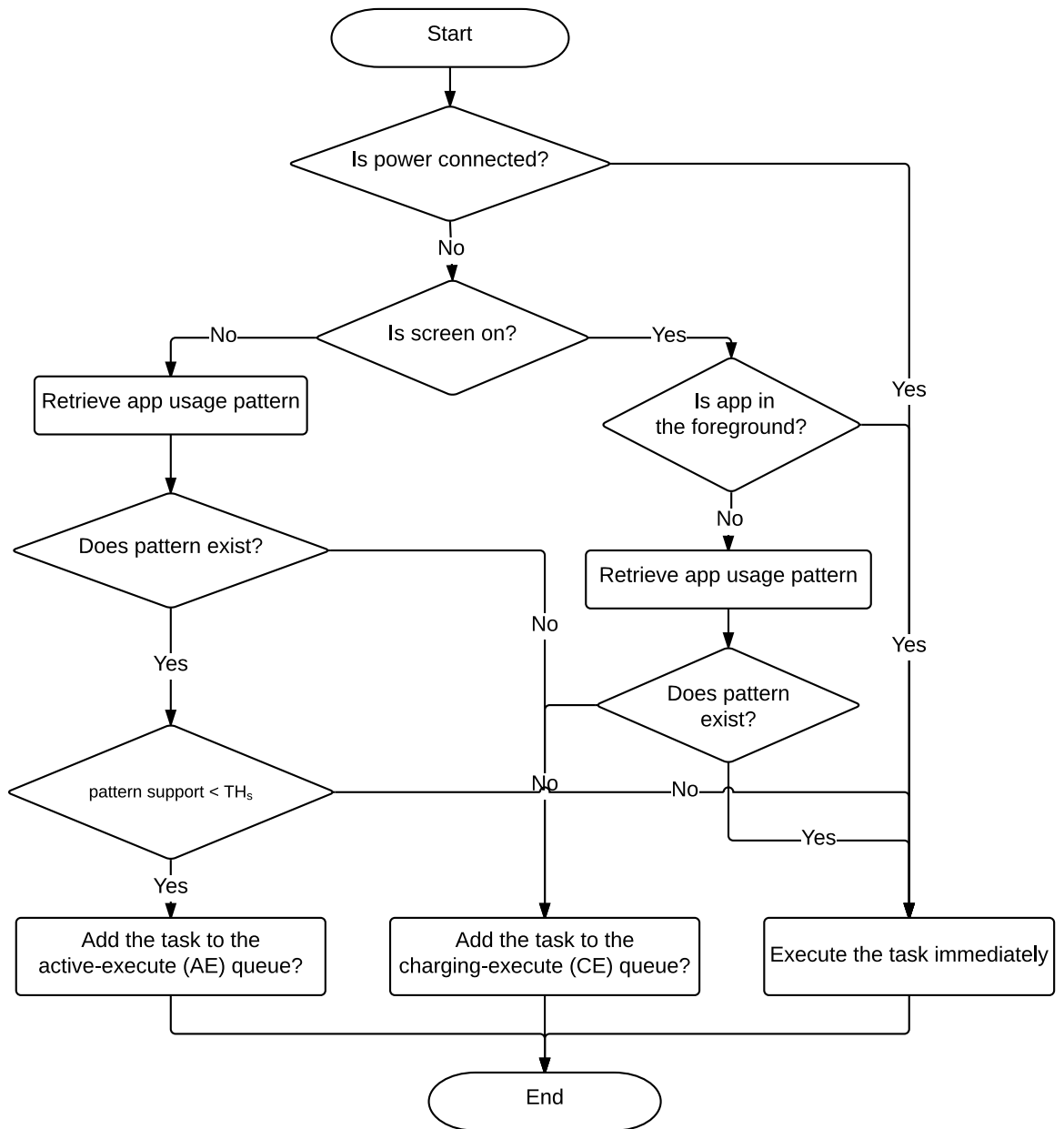


Figure 5.3: The flowchart of the UCASS strategy.

- *POWER_CONNECTED*: It schedules the tasks in both the AE queue and the CE queue to execute.
- *POWER_DISCONNECTED*: It stops to schedule the tasks in the CE queue and stops to execute the tasks in the AE queue if screen is off.
- *USAGE_PATTERN_UPDATE*: It retrieves the pattern items from the new usage pattern based on the current active period and location, and then reschedules the tasks in AE and CE queue based on the retrieved pattern items.
- *ACTIVE_PERIOD_UPDATE*: It reschedules the tasks in the AE queue and CE queue.
- *LOCATION_UPDATE*: Also, it reschedules the tasks in the AE queue and CE queue.
- *APP_STATE*: It schedules this task based on the scheduling policy we defined in the flowchart 5.3.

5.4 Implementation

We implemented the *UPS* system in Android 4.4.2 as a new *Service*¹ of the application framework layer. This section describes the implementation in detail.

5.4.1 UPS Power Manager Service

The same as other *Services* of the Android system, the *UPS power manager Service* includes a framework layer interface (defined in the *UPSPowerManager* class) and a background service (implemented as the *UPSPowerManagerService* class). We defined the framework layer APIs in an AIDL (Android Interface Definition Language) file and implemented the real operations in the *UPSPowerManagerService* class. When the Android system is booting, *SystemServer*² creates an instance of the *UPSPowerManagerService* class and registers it to the Android system with the service name “*ups_pm*”. Other *Services* or applications can get a

¹In Android system, a *Service* is an application component that can perform long-running operations in the background.

²SystemServer is a special framework layer Service of Android system. It manages other Services.

reference of this *Service* with the registered service name. After the system is booted, the *SystemServer* invokes the *systemReady* method of the UPS power manager Service to initialize it. In this method, it creates the user behavior monitor, the system event listener, and a worker thread. The user behavior monitor and system event listener are broad receivers³. The worker thread handles heavy tasks in a different thread.

The system event listener listens to system events such as *SCREEN_ON*, *SCREEN_OFF*, *POWER_CONNECTED*, etc.. Moreover, it listens to the *UPS* system generated internal events, such as *USAGE_PATTERN_UPDATE*, *LOCATION_UPDATE*, etc.. Whenever it receives an event, it iterates the registered energy-saving strategies in the *updatePowerSavingStrategies* method. In this method, the *listensToEvent* method of the strategies is invoked to check if the strategy listens to this event. If the result is true, it invokes the *update* method of the strategy to update its state based on the received event. The system event listener also checks if we should schedule the user behavior analysis task when receives the *POWER_CONNECTED* event. The *shouldScheduleUBAnalysis* method of the *SchedulePolicyProvider* class is invoked to perform this operation. If the result is true, it invokes the *performAnalyzation* method of the *UserBehaviorAnalyzer* class to start the analysis task.

The worker thread is hooked up with the message queue of the Android system. It communicates with the other parts of the UPS power management system through asynchronous messages. The worker thread runs in an independent thread; thus it can be used to process heavy tasks such as loading usage pattern.

5.4.2 User Behavior Monitoring

We modified the Android system to fire the *APP_STATE* event when an application switches to foreground or background. As we know that an Android application is composed by a group of *Activities*⁴. We tracked the life cycle of *Activities* to monitor when an application goes

³In Android system, *BroadcastReceiver* is a basic component that listens to system events.

⁴Activity is a basic component of applications in Android system. Each *Activity* class creates a window, which could be a full-screen window, an embedded window or a floating window.

to foreground or background. When a user-app interaction occurs, it invokes the *LocalLite* API to get the current location and fires an *APP_STATE* event. This event includes application state (*RESUME* or *PAUSE*), application id, application package name, location and time. We modified the *resumeTopActivityLocked* and *completePauseLocked* methods of the *ActivityStack* class, and the *startSpecificActivityLocked* and *startActivityUncheckedLocked* methods of the *ActivityStackSupervisor* class to fire the event.

The user behavior monitor starts with the UPS power manager service, and is registered to listen to the *APP_STATE* event. Whenever it receives an event, it retrieves the information about the user-app interaction and saves it to the *appusage* table of the database.

5.4.3 User Behavior Analyzer

In the user behavior analyzer, the statistical and data mining algorithms we described in Chapter 4 are implemented. Because user behavior analysis is a CPU intensive task, we encapsulate the computation in an *AsyncTask* class, which runs in an independent background thread. Before starting the analysis task, it loads user-app interactions from the *appusage* table, and also loads the parameters used by the analysis algorithms from the *SharedPreferences* data storage of the UPS power manager service. These parameters can be set by the user through the *setAnalyzerParameter* API of the UPS power manager. Then, it follows the three analysis steps we discussed in Chapter 4 to process the inputs and generate the usage pattern. After this process is done, it sends a *USAGE_PATTERN_UPDATE* Intent message to the UPS power manager and notifies it to load the new usage pattern from the database.

5.4.4 UCASS

We implemented this strategy into the Android system and integrated it with the UPS system. The strategy class maintains two queues, the active-execute (AE) queue and the charging-execute (CE) queue, to store the information of the tasks to be scheduled as well as the Intent message to start the task. The strategy listens to the events that we listed in Section 5.3.5, then the UPS power manager service notifies it when the related messages are received. If the

device is connected to the power source, it starts to execute the tasks both in the CE queue and the AE queue. If the device changes to active state, it starts to execute the tasks scheduled in the AE queue.

To implement this strategy, we need to intercept the starting procedure of Services, such that the activity manager talks to the UPS power manager service to decide how to handle the starting service request. We added the *startService()* method in the *IUPSPowerManager* interface, and added the *realStartService()* to the *IActivityManager* interface. The starting service work done in the original *startService()* method of activity method is moved to the new *realStartService()* method, implemented in the *ActivityManagerNative* and *ActivityManagerService* classes. In the *startService()* method of activity manager, it invokes the *startService()* method of the UPS power manager to handle the request. The *UPSPowerManagerService* sends a *START_SERVICE* event to the UCASS strategy. Based on the previously defined rules, the UCASS strategy decides when to invoke the *realStartService()* method of the power manager to really start the service.

5.5 Experiment Method

To understand user behavior and evaluate energy-saving strategies we collected real device usage traces from 14 volunteers. With these monitored data, we analyzed the usage patterns of the users. Also, we developed a system simulator and used these datasets to simulate the power consumption in different scenarios. In this section, we describe the experiment platform, data collection process and simulation method in detail.

5.5.1 Experiment Platform

Because the Android system is an open source mobile operating system, we can easily customize it to collect the data we want and implement our user-centric power management system. Thus, we modified Android 4.4.2 to monitor user-app interactions, battery power, process status, application status and system events. Then, we flashed the customized Android system to two *Nexus 4* smartphones. We chose this smartphone because its power supply

module integrates with current and voltage sensors. With these two sensors, we can accurately calculate the power dissipation of the device without instrumental tools, so that we can monitor the power of the device during the daily usage. Besides, we also calibrated the power models discussed in the Chapter 3 specifically for this device.

5.5.2 Data Collecting

We collected data from 14 volunteers. The monitoring period for each user is about a week. To accurately collect the regular behavior of these users, we asked each of them to reset the device and install nearly all the applications that they usually use on their own devices before the data collecting procedure starts. After a user finished the experiment, we retrieved the data from the device and then gave it the next user to use.

In the log files, user-app interaction records include application uid, package name, time, location and operation type. A process activity record contains the time, active status, process id, CPU time used and the corresponding application uid. The system events we monitored include wakelock usage, location service usage, power supply state, screen state and so on. Also, we collected the state change of hardware components and resource utilization to estimate the power. With these data, we can not only analyze the usage pattern of users but also simulate the power management of the Android system to evaluate energy-saving strategies.

5.5.3 Trace-driven Simulation

Since it is impossible to generate exactly the same user activities in two different periods, we use the simulation method to compare the energy consumption of different scenarios. We developed a simulator to compare the power consumption of the system in different scenarios. For example, the simulator can estimate the energy consumption while enabling the UCASS strategy, and compares the result with the normal scenario. The simulator takes the log files, and usage pattern analyzed from user-app interactions as inputs.

In the experiment, we assume the usage pattern does not change. We analyzed the usage pattern of each user from their monitored dataset first and then applied the usage pattern

to the simulation of the same dataset. In the real implementation, the usage pattern is analyzed based on previous user-app interactions. Most users' usage pattern does not change very frequently. Thus, we can safely make this assumption. Then we simulate the system with different strategies were enabled and different system configuration. During the simulation, the system component models estimate the power of applications based on the power models we introduced in Chapter 3, and calculate the energy saving of the strategies.

5.5.4 Simulator

We built an event-driven simulator to compare the energy consumption of the system in different scenarios. The simulator simulates system events, application activities and power manager of the system. With this simulator, we can setup the system model with different power managers and different energy-saving strategies. In this section, we relate the design and implementation of the simulator.

Simulator Design

Because we focus only on the time when some special events occur, we designed the simulator as an event-driven system. The event we focused on can be an user-app interaction, a change of system status, a wakelock request, a battery power value, a process state, etc. Each line of the log files corresponds to an event. During the simulation, we read the log files line by line and parse each line into event. These events trigger the simulator to move to new states until the simulation finishes. Finally, the simulator generates the simulation result.

As shown of Figure 5.4, the simulator includes two parts: the simulator framework and the models. The simulator framework includes four components: configuration loader, log parser, event listener manager and report generator. The configuration loader loads configuration files, such as usage pattern and simulation configuration before simulation starts and initializes the status of the models (e.g. registering some models as event listeners). Then the log parser parses the log files and converts each line into an event. It gives the events to the event listener

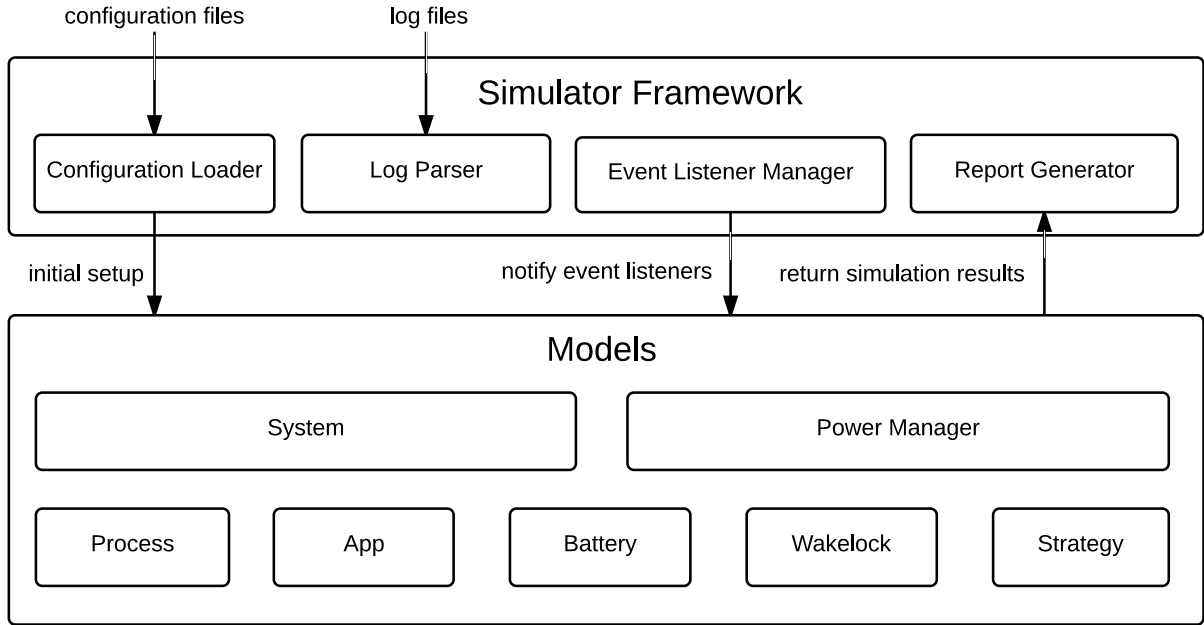


Figure 5.4: The architecture of the trace-driven simulator.

manager, which sends the event to the registered event listeners. After the file is parsed, the report generator generates the simulation result.

The models are designed to simulate the behavior of some core parts that are related to the power management of the mobile operating system. In our simulator, we build power models for the system, power manager, process, application, battery, wakelock, location service and energy-saving strategies, shown as Figure 5.4. The top level models, such as system model and power manager model, implement the *EventListener* interface, such that they can receive the event notifications. Then, they propagate the events to low-level models. Whenever a new event is received, the models that listen to this event update their state accordingly. At the same time, the battery model evaluate the power consumption of the device. The strategy model is implemented with the power-saving strategies. We set up which strategy to use before the simulation starts.

5.6 Evaluation

In the simulator, we implemented the **UPS power manager** and the *UCASS* strategy. During the simulation, we did not consider the change of system applications after rescheduling

non-system applications to other periods because we can not tell how much the system application power will decrease. This will be part of our future work. In this section, we compare the power consumption of these two scenarios:

Scenario 1 The *UPS* system as well as the *UCASS* strategy are disabled.

Scenario 2 The *UPS* system as well as the *UCASS* strategy are enabled.

In Chapter 2, we had already analyzed the device and application power consumption in Scenario 1. We found non-system applications generates a considerable amount of activities while running in the background. Most of these activities are not important to the user, and we can use the *UCASS* strategy to reschedule these background activities when the device is charging or active. Such that, we can decrease the battery power consumption caused by background application, which has no usage pattern during different active periods, generated activities. In this section, we mainly analyze the power consumption in Scenario 2.

5.6.1 Power Consumption Analysis

Figure 5.5 shows the percent of saved energy when we enable the *UCASS* strategy. For all the users, this energy-saving strategy can save more than 18 percent of battery energy. For some of users, the energy saving is more than 30 percent. We use the average active power in Table 5.1 to calculate the extended battery life in Scenario 2. Except user 4, all the other user's battery life can be saved for more than 15 minutes; the average extended battery life is about 25.34 minutes. The result shows the extended battery life when the device is active. The extended battery life is still considerable when considering the high active state power consumption of the device.

To further understand which applications' (non-system applications) power consumption can be saved, we use the experiment result of user 15 as an example to analyze. From Table 5.2, we can see that most top energy consumers' (non-system applications only) power consumption can be greatly saved. The power consumption of applications, such as Alipay

Table 5.1: The average power and the length of the experiment period of users.

User ID	Experiment Period (days)	Avg Active Power (mw)	Avg Idle Power (mw)
1	9.3	1078.93	425.76
2	8.13	543.88	358.9
3	8.67	1273.95	372.1
4	9.37	1071.88	401.98
5	7.04	1178.69	464.48
6	7.87	1075.26	453.01
7	6.97	959.51	437.09
8	7.95	1095.66	496.37
9	5.14	1287.66	375.94
10	11.02	1288.12	419.49
11	12.91	1253.41	392.86
12	16.82	1094.75	233.77
13	15.5	1012.79	335.55
14	7.67	981.66	315.93

and Amazon, can be saved for 100%, that is mainly because the user never used these two applications. However, they still consume a large amount of battery energy, their activities can obviously be sieved or rescheduled without influence user experience. Google Play is a background service that update application information, the developer should optimize the behavior of this application to make it more energy efficient. The energy saving of *Chrome* and *Youtube* is less than 13% because they most of the activities were generated when they were in the foreground. For this kind of applications, we still need to use energy-adaptation techniques to make them more energy-efficient. For example, we can disable image on the webpages for Chrome in the energy adaptation state.

5.7 Related Work

In this section, we talk about the related work from two areas: power management through controlling application states and user-centric system design.

5.7.1 Power Management through Controlling Application States

Traditional power management systems become less effective if users install more mobile applications. That is mainly because the redundant application activities cause the system

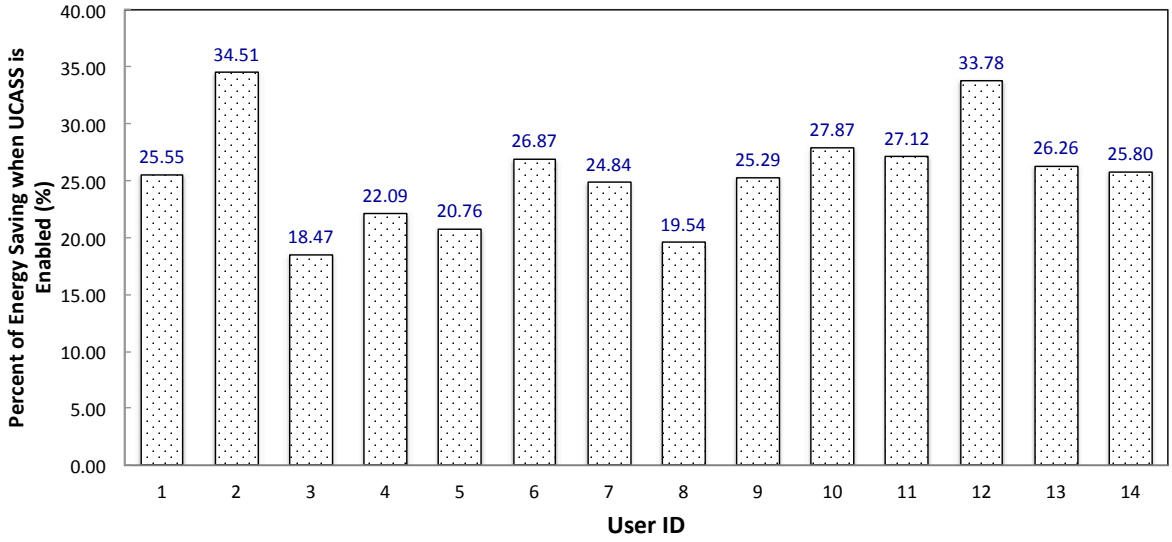


Figure 5.5: The percentage of energy saving when the UCASS strategy is used.

keeps active most of the time. This new situation hastened a group of new power management systems [43, 44] that aim to limit the power usage of applications. Zeng *et al.* designed and implemented a new power management system called *EcoSystem*, which considers battery energy as one kind of system resource in their proposed *Currentcy Model* [43]. Currentcy is the common unit for energy accounting and allocating. In each epoch, an application can only consume the allocated amount of energy. By limiting the available energy in each epoch, the system can guarantee the predefined battery life. Similar with *EcoSystem*, Roy *et al.* proposed a hierarchical pattern to control the energy consumption of applications, and implemented the idea into the *Cinder* operating system [44]. Moreover, they added the idea of cooperation, which means applications that use the same hardware device can collaborate to active the device.

In [1], Flinn *et al.* used the energy adaptation technique to balance battery life and application quality. They argued that applications should dynamically modify their behavior to conserve energy and satisfy the battery life requirement of users. Their experiment result shows the potential to save battery life through energy adaptation is about 30%. In our previous work, we also designed an energy adaptation framework for energy-aware application development. Park *et al.* proposed a new scheme to reduce non-critical alarms and decrease energy con-

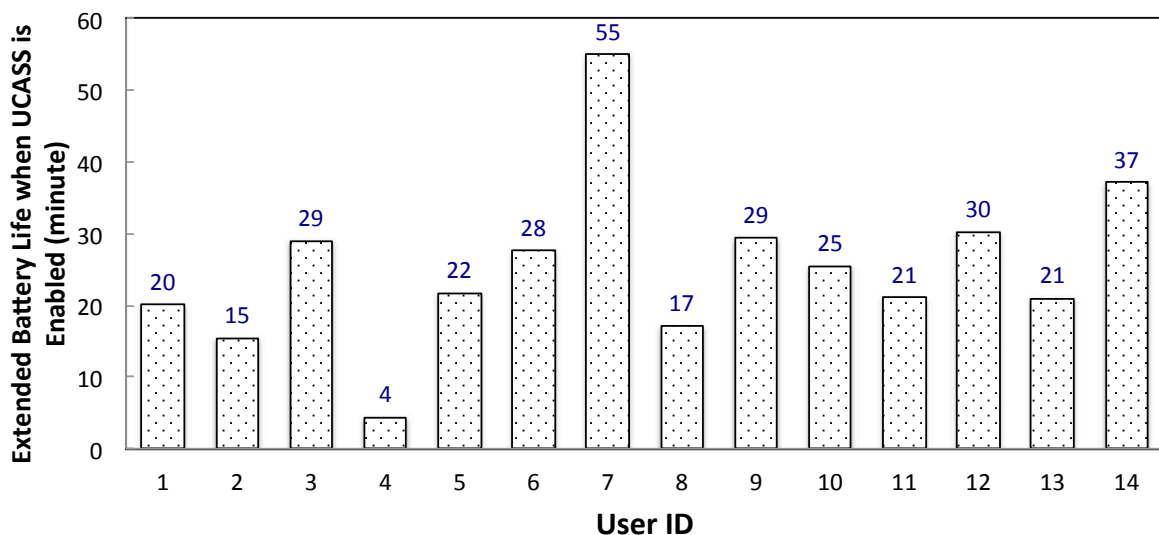


Figure 5.6: The extended battery life when the UCASS strategy is used.

sumed by applications that generate alarms [27]. In [45], Elmalaki proposed to schedule part of application activities to the charging period to save battery energy consumed by applications.

In [86], Chen *et al.* also noticed the improper energy usage of background applications. They proposed a method to identify unnecessary background activities and suppress them during the idle state. In [24, 87], Bolla *et al.* proposed ASP (application state proxy) to suppress applications on smartphones. Their method only restart the background activity if a new message arrivals.

5.7.2 User-Centric System Design

These methods, however, fail to balance energy-saving and user experience. A new trend is using user behavior to design energy-saving strategies. User behavior is not only helpful for eliminating redundant application activities but also useful for optimizing traditional power managers.

The significant diversity between user behaviors implies traditional system mechanisms, such as power management system, that work for the average case may not be effective for most users [64]. Falaki *et al.* introduced a personalized energy drain predictor and found that learning and adapting user behavior is more effective than traditional methods. They proposed

Table 5.2: The power consumption of non-system applications of user 14 in Scenario 1 and 2.

Application	Power Consumption (mj)		Energy Saving (%)
	Scenario 1	Scenario 2	
Baidu	4414036.79	1353909.64	66.93
Youtube	4403636.81	3834274.98	12.93
Tencent News	3214128.10	1622536.52	49.52
Chrome	1352400.61	1197871.21	11.43
Alipay	1201248.42	0	100.00
Google Play	1109439.46	0	100.00
QQ	902248.45	30159.52	96.66
Amazon	785164.31	0	100.00
WeChat	675128.71	100669.99	85.09
QQ Sports	627178.51	411109.01	34.45
Others	5498340.37	479659.53	91.28

to add a light-weight user behavior monitor to existing mobile operating systems. Lin *et al.* proposed an optimized location service named A-loc [88], which uses accuracy models to estimate the location accuracy requirement of mobile applications. Then, it selects the most energy-efficient location sensors that can guarantee this accuracy requirement.

Balasubramanian *et al.* studied the power consumption of 3G, GSM and WiFi, and proposed TailEnder to reduce the tail energy consumption of mobile applications [51]. They used the task grouping technique to reschedule delay-tolerant applications' network request, such as e-mail. In [89], Vishal *et al.* propose to schedule network access tasks based on predicted user activity. Khairy *et al.* proposed the "Smartphone Energizer" [90] to learn user behavior in various contextual situation and offload computation to the server side.

5.8 Summary

In this section, we describe the design and implementation of the *UPS* power management system and the *UCASS* energy-saving strategy in detail. We used simulation method to compare the energy consumption of different scenarios. The experiment shows more than 18% of energy can be saved after using the *UCASS* strategy and on average the battery life can be extended for about 25.34 minutes.

With user's usage pattern, many aspects of the system should be revisited for energy efficiency. Many system services can be optimized. In the next two chapters, we present our solution for optimizing the power consumption of location service and wakelock mechanism.

CHAPTER 6

LOCALITE : AN ENERGY-EFFICIENT LOCATION PROVIDER

Nearly all modern smartphones supply location service, which uses some specific hardware components (GPS, wireless network, and cellular network) to pinpoint user location. Location service enables mobile applications to provide various kinds of useful functionalities to users. However, our monitored data of location service usage shows the API of location service was redundantly used and thus can be optimized for energy-efficiency.

6.1 Introduction

Modern smartphones, which are characterized by the powerful computing capability and diverse functionalities, are totally different with traditional cell phones. With our smartphones, we can surf web pages, take pictures, record videos, share personal information with friends via social networks and navigate when driving. The usage pattern of mobile devices is also significantly changed because the vast functionality difference. Nowadays, the smartphone is more like a personal assistant, with which we acquire various information and keep in touch with others. Generally speaking, we use our mobile phones much more frequently than before.

To support the various new functionalities, current smartphones are usually equipped with much more hardware components than before. However, some of these new hardware components are very power hungry. We cannot equip the smartphone with a large enough battery to supply a long enough battery lifetime because the size limitation of the mobile device. In fact, different people have different battery lifetime expectations, and it is hard to please everyone. Extending battery life will always be one of the main considerations when we design various aspects of mobile devices. Based on the result of user behavior analysis, we found the user behavior of most users is location-dependent, which means a large amount of location requests that started at the same location were redundant.

We did a statistical experiment to stat how many location requests the device generated in daily use. Users report their location service usage through a web page whenever the applica-

tion made an observable location request. The result shows that the reported location service usage ranges from 10 to 71 times per week. However, we found the logged location requests are much more than the user observed. We propose *LocaLite* to optimize the energy consumption of location service from user-centric point of view. *LocaLite* exactly follows this train of thoughts to eliminate the repeated location requests through caching earlier received locations. The design of *LocaLite* naturally embodies the location-dependent characteristic of user behavior. Most end users only use their smartphones in several locations, such that many location requests initiated at the same place are, in fact, a waste of battery energy.

6.2 Background

In this chapter, we describe the background information about location service, and analyze how it is used by applications.

6.2.1 Location Service

Location service is widely used by mobile applications for various kinds of purposes. For example, weather applications and local service applications use the present location of the user to find context-related information, and social networking applications share user location to their friends. Location service relies on the hardware components, such as cellular, WiFi, or GPS, to triangulate the current location of the mobile device. These hardware components are usually very power hungry. We measured the power dissipation of location service on Nexus 4 when using location service in different ways. On average, the power requirement of acquiring location with GPS, network and fused provider is 332.06 milliwatts, 312.82 milliwatts and 408.76 milliwatts respectively.

Since different kinds of applications have different requirements, mobile platforms, such as the iOS platform and the Android platform, usually support two default location providers: GPS location provider and network location provider. The GPS location provider uses GPS to determine the location; the network location provider uses cellular radio and WiFi to acquire the location. These two location providers are power-hungry because they directly use

hardware components to acquire the information.

The Android system supplies two other location providers [91]: fused location provider and passive location provider. They encapsulate the default location providers and make location service easier to use. The fused location provider chooses the suitable location provider based on the accuracy requirement. The passive location provider returns the location received by other applications. The passive location provider nearly consumes no power, but the location result is inaccurate if other applications did not request location recently. Similarly, the iOS platform [92] supports two location services: standard location service and significant-change location service. For the standard location service, we can configure the desired accuracy of the location data and the minimum distance before updating a new location. Similar with the fused location provider of Android, it utilizes all available hardware components to acquire the location. The significant-change location service uses cellular radio to determine the location of the mobile device. Thus, it dissipates much less power than the standard location service.

6.2.2 Usage of Location Service

Applications use the location service for several purposes: navigation, tracking the movement, labeling the current position, tagging the content to publish, searching local businesses, retrieving location-related information and so on. Among all these usage scenarios, only navigation and tracking the movement requires continuous location updates. Other usage scenarios, however, only need to query the location for just once, and most of them only need a coarse location.

However, we found that applications aggressively use location service. Figure 6.1 shows the time serials of the location requests (passive excluded) in about five days. The x-axis of the figure shows the time in seconds, and each vertical line shows the period of location requests. From this figure, we can see that *Amazon client* (amazon), *Google Play Service* (gms) and *Weather* (weather) used location service very aggressively. Besides, improper usage of location service is also common. For example, *Twitter* requests user's current location whenever the

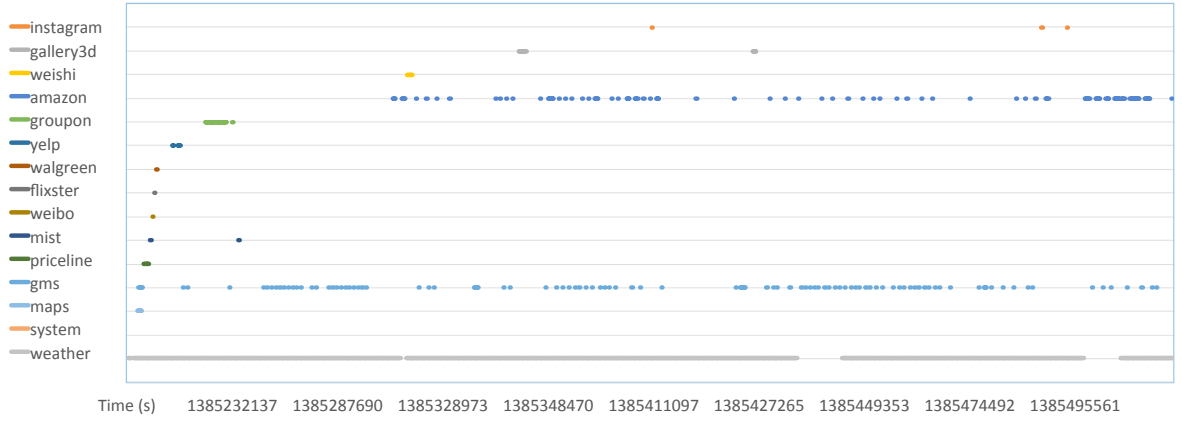


Figure 6.1: The time serials of the usage of location service in about 5 days (passive requests are not included).

application is switched to the foreground and stopped when it received the location. Foremost, most applications use location service to get the accurate location, which is not necessary. Table 6.1 lists the location providers used by several popular applications.

6.2.3 Location Requests Analysis

From the usage traces of 14 users, we found the number of location requests is significantly different between users, as shown of Figure 6.2. User 1 and user 10 generated much more location requests than other users during the experiment. User 10, user 11, user 14 and user 15 generate more GPS-based location requests. User 1, user 3, user 5, user 6, user 8 and user 9 generate more network-based location requests. User 4 and user 7 generated much fewer location requests than others in the experiment. That is not only because they use the device less frequently than others, but also because they disabled several applications (*Google Play Service* and *Google News & Weather*) that generate a large amount of location requests.

We use the result of user 10 to analyze which application generates the location requests. From Figure 6.3, we found *Google News and Weather* and *Google Play* generated much more location requests than other applications. Their location requests account for about 80.43 percent. Obviously, these two applications are not well-designed. Most of their location requests are redundant. *Fengyun* and *Youku* are video players, they are not supposed to collect user loca-

Application	Used Providers
weather	passive, network
google maps	gps, network
priceline	gps, network, passive
sina weibo	gps
flixbus	network
walgreen	gps
yelp	gps, network, passive
gms	passive, gps
groupon	fused
amazon	network, gps
Tencent weishi	gps
instagram	network, gps, passive
gallery3d	gps, network
system	passive

Table 6.1: The location providers used by several popular Android applications.

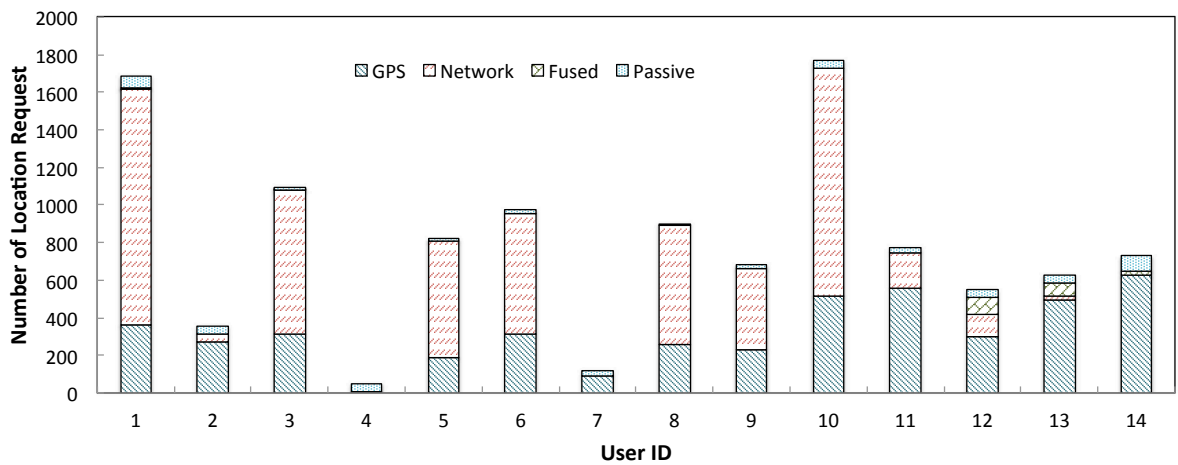


Figure 6.2: The number of location requests of users.

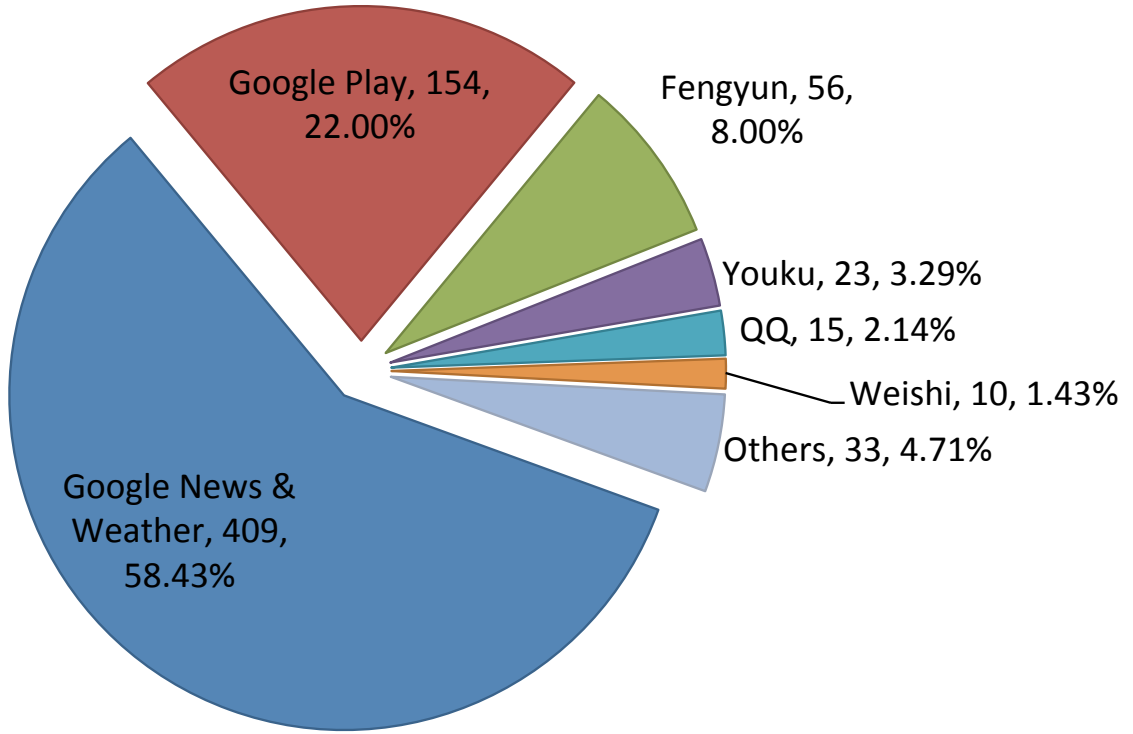


Figure 6.3: The percentage of application generated location requests of user 10.

tions. *QQ* and *Weishi* are social network applications, they only need coarse-grained location. From this analysis, we found the huge potential to save battery energy consumed by location service.

6.3 LocalLite

In the design of the *UPS* system, we record the location information of user-app interactions. Existing location providers, however, are very power hungry. Using them will significantly increase the energy consumption overhead of the *UPS* system. To solve the problem, we propose a light-weight location provider - *LocalLite*.

6.3.1 Design Considerations

We propose *LocalLite* because of several reasons. First, the *UPS* system and most mobile applications only need coarse location information. Besides, the location distribution of user-app interactions shows most users' activities occurred at a very limited number of loca-

tions, and in most of these locations the devices were connected to WiFi. This observation shows most location requests are duplicated, and much energy can be saved from eliminating the unnecessary location requests. The location-dependent pattern of user behavior was also observed in the previous publications [34, 35]. The design of *LocaLite* naturally embodies the location-dependent characteristic of user behavior.

LocaLite eliminates the repeated location requests through caching earlier received locations. It uses the information of wireless router to mark each location because most of the wireless routers are immovable. Whenever the system receives a new location request, *LocalLite* checks whether the device is connected to a wireless router and whether the corresponding location related with the current network environment exists. If the location exists in the cache, it returns the location and stops the location request. Otherwise, other location providers will be used to retrieve the user's location, and *LocalLite* caches the new location when it is received. Finally, we set an expiration time to each record in case some wireless access points were moved. By default, we set the expiration time to seven days. Expired records will be periodically deleted.

6.3.2 Location Caching

The user behavior of one user may be significantly different with other users; the number of locations cached on mobile devices varies from several to dozens. To improve the performance of location retrieving, we designed a two-level location cache structure to store WiFi and location information. The first level uses local storage to store all recorded information, and the second level uses memory to store several most recently visited locations. This design also naturally utilizes the user behavior that most users frequently request the same location consecutively, and most users only visit a very limited number of places in a short time. Thus, most of the time, we can find the location from the memory-level cache.

If a location request cannot find the corresponding location with *LocalLite* or when WiFi is not available, a normal location request will be generated. When the system receives the

requested new location, it will check whether the device is connected to WiFi. If WiFi information is available, it saves WiFi information and location to storage and memory-level cache. We use the least recently used algorithm (LRU) [93] to manage the location records of the memory-level cache. When a new location is received, it also needs to run the LRU algorithm to replace the least used item with the new location. When WiFi is not connected, *LocalLite* does not record the result. The pseudo code of location caching is shown as Listing 6.1.

Listing 6.1: The pseudo code of the location caching Algorithm.

```

ArrayList locCache ;
int cacheSize ;
void cacheLocation(Location loc , Wifi wifi){
    if(wifi.isAvailable() == false)
        return ;

    Location last = queryLocation(wifi);
    if(last == null ||
        last.getAccuracy() < loc.getAccuracy()){
        storeLocation(wifi , loc);
    }

    if(locCache.contains(loc)){
        locCache.remove(loc);
        locCache.insert(0, loc);
    }else if(locCache.size() >= cacheSize){
        locCache.remove(locCache.size() - 1);
    }
    locCache.insert(0, loc);
}

```

```
}
```

6.3.3 Location Retrieving

LocalLite uses the information of the connected network to find the current location. Although it's not as accurate as other location providers, it returns locations that can satisfy most applications' requirement without consuming too much battery energy. As shown of Listing 6.2, when WiFi is not available, *LocalLite* will use other location providers to request the location information. Otherwise, it first searches the memory-level cache. If the corresponding location information cannot be found from the memory-level cache, it then searches the location from local storage. The location will be returned if the location was saved in either memory-level cache or local storage-level cache. Otherwise, a location will be requested through other location providers.

Listing 6.2: The pseudo code of the locatin retrieving algorithm.

```
ArrayList locCache ;
int cacheSize ;
Location retrieveLocationOrStartUpdate (Wifi wifi){
    if (wifi.isAvailable() == false){
        updateLocation (wifi);
        return null;
    }
    //query from memory cache
    Location loc =
        findCachedLocation(new Location(wifi.getBssid()));
    if (loc != null){
        locCache.remove(loc);
        locCache.insert(0, loc);
    }
}
```

```

        return loc;
    }
    //query from local storage
    loc = queryLocation(wifi);
    if(loc != null){
        if(locCache.size() >= cacheSize)
            locCache.remove(loc);
        locCache.insert(0, loc);
        return loc;
    }
    updateLocation(wifi);
    return null;
}

```

6.4 Implementation

We implemented the *LocalLite* location provider in Android 4.4.2. To understand how location service was used by applications, we modified the *requestLocationUpdatesLocked* method and the *removeLocationRequestLocked* method in the *LocationManagerService* class of Android to log location request/release information and WiFi information.

We use a SQLite database to save the location and related WiFi information, such as the basic service set identifier(BSSID), network id and the service set identifier (SSID). These operations of managing this database are encapsulated in the class *LocalLiteOpenHelper*. Besides, we use a list data structure to cache the most recently used locations. We added two functions, *storeLocation* and *retrieveLocation*, in the class *LocationManagerService*.

The *storeLocation* method is invoked by the method *handleLocationChanged* if the returned location is not passive. It follows the process we defined in Listing 6.1 to store the location with the methods of the class *LocalLiteOpenHelper* and update the memory-level lo-

cation cache. The *retrieveLocation* method is defined as public (we add this method to the *ILocationManager.aidl* interface definition file), and it will be invoked by the *retrieveOrUpdateLocation* method that we added to the class *LocationManager*.

6.5 Evaluation

The monitored datasets include the location request information and WiFi information. More specifically, we recorded the location provider, start time, end time, and application UID of each location request. In this section, we analyze how location service was used and simulate the energy consumption in the following two scenarios:

Scenario 1 All the applications use the location provider they used before.

Scenario 2 All the applications use the LocalLite location provider.

6.5.1 Energy Consumption Analysis

In the simulation, we calculated the energy consumed by location service in these two scenarios. We use equations 6.1 to calculate the energy consumption of each location request. The power values in the equation are the average power consumed by location service that we measured on the experiment device. Besides, we use the average device active power of each user, shown as Table 5.1, to calculate the battery life consumed by location service.

$$P(p_i) = \begin{cases} 312.82mw & p_i = \text{network} \\ 332.06mw & p_i = \text{gps} \\ 408.76mw & p_i = \text{network, gps} \end{cases} \quad (6.1)$$

$$E_{normal} = \sum_{i=1}^n P(p_i) \times t_i \quad (6.2)$$

Compared with Scenario 1, about 98.5 percent of battery energy consumed by location service can be saved on average in Scenario 2, as shown of Figure 6.4. Also, we found that applying *LocalLite* can save more than 90 percent of battery energy consumed by location service for all the 14 users. That is mainly because some applications generated many redundant

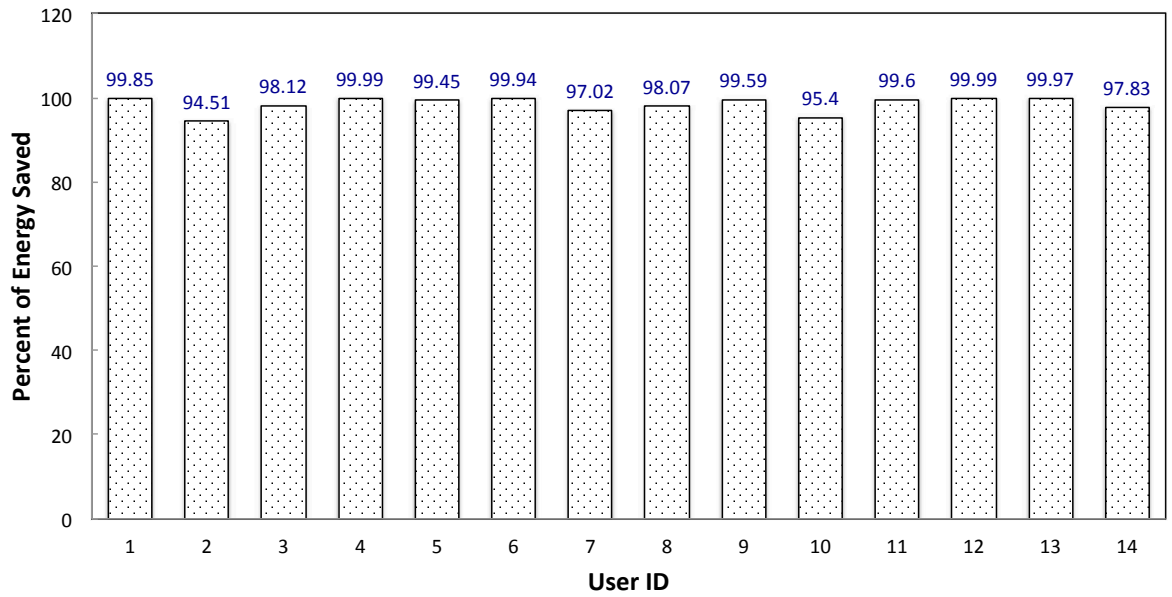


Figure 6.4: The percentage of energy saved in Scenario 2.

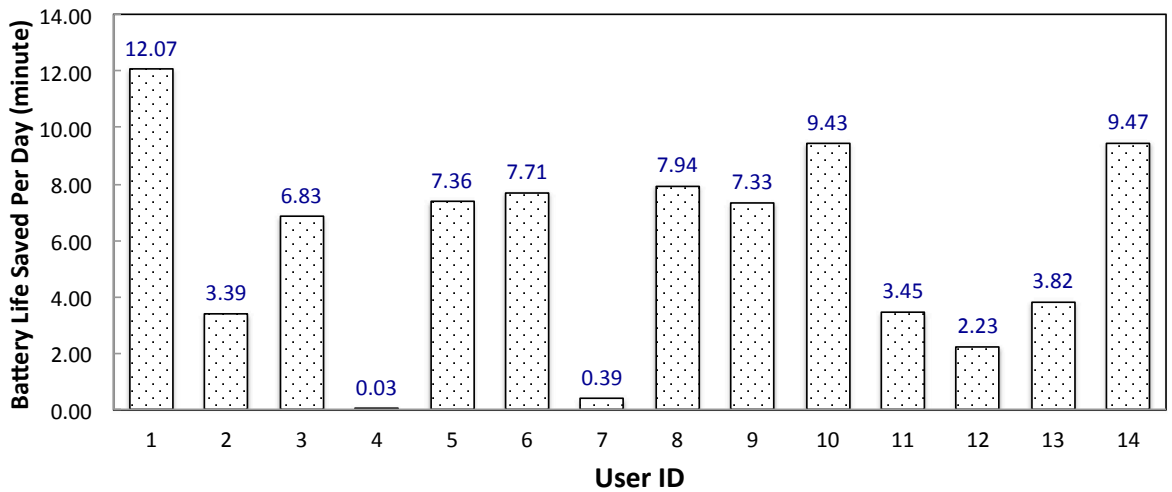


Figure 6.5: The battery life extended per day in Scenario 2.

location requests. The amount of energy saved by LocalLite is related with user behaviors. If a user usually uses his or her smartphone in environments that WiFi is available, more energy will be saved. On the contrary, if a user's most device usage is in outdoor environments (without WiFi), then less power can be saved.

Figure 6.5 shows the battery life saved per day after using the LocalLite location provider. Since location service only consumes about 3.13 percent of battery energy on average in the experiment of all users, the battery life extended is still considerable. Again, the amount of energy saving is greatly related to user behavior. From this analysis, we can see that *LocalLite* can effectively eliminate the energy waste caused by redundant location requests. This new service makes location-marked user behavior sensing becomes realistic in real usage.

6.6 Related Work

The main topic of this chapter is optimizing the energy consumption of location service. In this section, we discuss previous work about improving the energy-efficiency of location service.

In [94], Zhuang *et al.* proposed an adaptive location sensing framework. They use context information to find the best location sensing mechanism and detect the mobility of users with low-power sensors to suppress unnecessary location sensing. Lin *et al.* also designed an adaptive location service based on the observation that the requirement of location accuracy varies with location [88]. The service chooses low-power location providers based on the context information.

In [21], Liu *et al.* uses the computation offloading technique to optimize the GPS-based location service. With their solution, the device only needs to sense a very limited amount of raw GPS signal data and uploads the data to the server. The server side leverages information saved in GNSS satellite ephemeris and an Earth elevation database to derive good quality GPS locations from the raw signal data. Compared with the heavy signal processing on a standalone GPS receivers, the new solutions can achieve three orders of magnitude lower energy

consumption.

Huang *et al.* also found the redundant usage of location service in [95]. They developed E2A2 (energy efficient and accuracy aware), which uses group location to represent individual device location. Wang *et al.* enables continuous outdoor location sensing with WheelLoc [96], which captures user's user mobility trace first and then calculating location by time-aware and speed-aware sensors. It avoids power-hungry sensors and relies on low-power sensors to locate.

6.7 Summary

Location services that use GPS, wireless network or cellular network to pinpoint the location of users are power hungry. These services, however, are aggressively used and missed by applications. To improve the energy-efficiency of location service, and enable location-based user behavior sensing, we propose a light-weight location provider named *LocalLite*. *LocalLite* combines wireless network access points marked contextual information with the location to eliminate redundant location requests. Experiment result shows about 98.5 percent of battery energy consumed by the location service can be saved.

Except location service, there are many other system services of the mobile operating system can be optimized based on user's usage pattern. In the next Chapter, we optimize the energy-efficiency of the wakelock mechanism.

CHAPTER 7

WAKEFILTER : MAKE ANDROID FALL ASLEEP THROUGH WAKELOCK FILTERING

The Android system introduced the opportunistic suspending technique to optimize the power consumption of mobile devices. This technique uses wake blockers (wakelock) to keep the system active while executing important tasks. This mechanism, however, fails to work properly if wakelock is misused or abused by applications. In this chapter, we propose *Wake-Filter* to solve this problem and make the system work more in low-power mode.

7.1 Introduction

Traditional power management systems define idle as “CPU utilization is zero”. In the real case, however, CPU utilization never becomes zero. Android system introduces the opportunistic suspending technique [41, 42, 97], which redefines idle as “no important tasks to execute”. It uses wakelock or suspend blocker to keep the system active while executing significant tasks. Both kernel space module and user space applications can apply for wakelocks. When no wakelocks are held by the system, the entire system (except for the components that listen to wake-up events) will be suspended to memory, rather than put various system components into a low-power state [41]. When the system works in the suspend-to-memory state, only a very low power dissipation is required for refreshing memory and powering a few devices that can wake up the device. On the contrary, the whole system will keep active to process important tasks. This technique is helpful to mobile devices. The system can quickly wake up and resume to active state to handle user interactions or hardware events.

In Android system, any application can apply for different types of wakelocks. Table 7.1 shows the available wakelock types in Android and the status of related device components. Partial wakelock is a special wakelock that keeps CPU active even after the user presses the power button or display timeouts. Listing 7.1 shows an example of wakelock usage. Normally, applications should release the wakelock they requested as soon as the task finishes. However,

wakelocks are usually not released on time or even failed to be released. Vekris *et al.* found that some applications never release wakelocks because the program throws exceptions while executing the important task, and the program does not handle the exceptions properly [37]. In [38, 39, 40, 98, 99], wakelock misuse is also found by researchers from different research groups. Except that, we found that wakelock abuse is as well common in Android system based on the device usage traces of 14 users. We will talk more about our analysis result in Section 7.2.

Listing 7.1: The example code of wakelock usage.

```
PowerManager pm = (PowerManager) getSystemService (POWER_SERVICE);
WakeLock wakeLock = powerManager.newWakeLock(
    PowerManager.PARTIAL_WAKE_LOCK, "WakelockTag");
wakeLock.acquire();
execute_the_important_task();
wakelock.release();
```

Table 7.1: The available types of wakelocks in Android, and the corresponding status of hardware components when different type of wakelocks are held.

Wakelock Type	CPU	Screen	Keyboard
PARTIAL_WAKE_LOCK	ON	OFF	OFF
SCREEN_DIM_WAKE_LOCK	ON	DIM	OFF
SCREEN_BRIGHT_WAKE_LOCK	ON	BRIGHT	OFF
FULL_WAKE_LOCK	ON	BRIGHT	BRIGHT

As more and more third-party applications are developed, wakelock misuse and abuse will become even more common in Android system. That is mainly because not all the developers are experienced, many mobile applications are poorly-written. To solve the problem, Kim *et al.* [39] proposed PR-wakelock (Predict & self-Release Wakelock), to predict the misuse of wakelocks and forcibly release these wakelocks. In [40], Alam *et al.* proposed a data flow based analysis strategy to determine the placement of wakelock statements. Different with

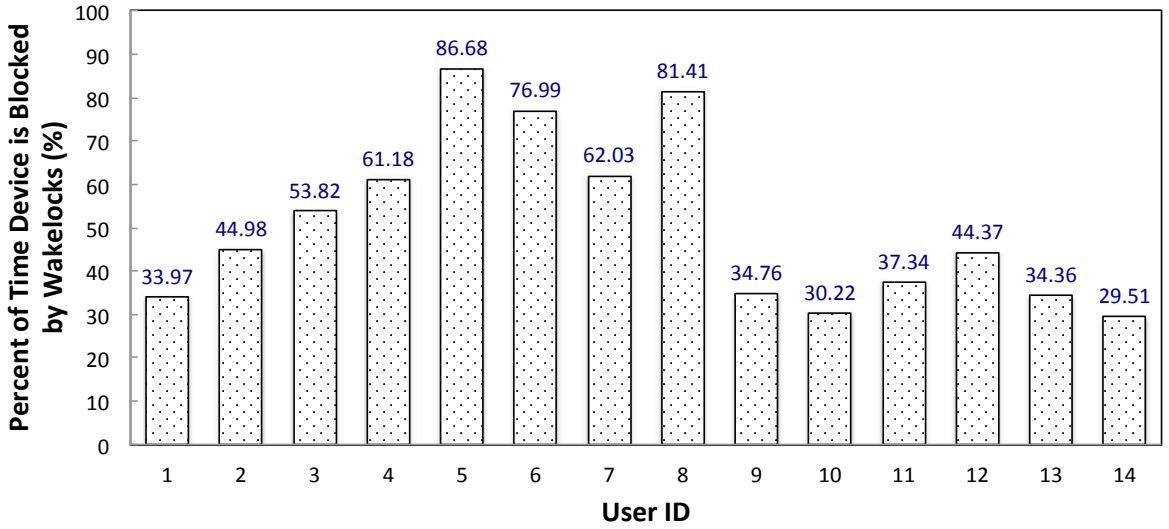


Figure 7.1: The percentage of time that the devices were blocked by wakelocks in the idle state.

these methods, we solve the problem from an unusual direction. We combine the idea of energy adaptation and user-centric system design to intelligently filter wakelocks created by insignificant applications. Based on the analysis result of user behavior in Chapter 4, we know that an “important” task may become an “unimportant” task at different context. We propose the *WakeFilter* mechanism to detect wakelock misuse and abuse based on the usage pattern of each user, and forbid the unimportant tasks to keep the system active in the device-idle state. Such that, the device can work more in the low-power state.

7.2 Wakelock Usage Analysis

When we collected the device usage traces of users, described in Chapter 5, we also saved the information of the wakelock request and wakelock release events. In this section, we analyze these wakelock usage traces. Because wakelock misuse and abuse only waste battery energy when the device works in the device-idle state (screen off), we only consider wakelock events generated when the device is idle. For wakelocks that across both the idle state and the active state, only the part of time in device-idle state will be considered.

7.3 How Long Devices Are Blocked by Wakelocks?

We analyzed how long the devices were blocked to sleep during the whole experiment period. For all the users, greater than 30% of time, the devices were blocked by wakelocks in the device-idle state, and the average value is 50.83%, as illustrated in Figure 7.1. In some serious case (user 5 and user 8), the devices were kept active by wakelocks for more than 80% of time. This result shows applications aggressively used wakelocks, and the opportunistic suspending technique cannot effectively save battery energy. The situation becomes even worse if users install more applications on their mobile devices.

7.3.1 Which Applications Blocked Device to Sleep?

To understand whether these wakelocks are necessary to the system, we use the result of user 6 to analyze which applications generate these wakelocks. From Table 7.2, we found *android*, *WeChat* and *Google Play Service* generated much more wakelocks than any other applications. Their wakelocks account for about 93.28% of the total time that the device is blocked (device-blocked time), shown as Figure 7.2. We know that system applications, such as *android* (uid is 1000), includes many system services, which run in the application framework layer of Android system to support non-system applications (their uid is larger than 10000). If the activities of non-system applications decrease, system applications' activities will also decrease. WeChat is an instant message application, and it runs in the background to periodically synchronize with the server. This application, however, generates wakelocks too frequently. Besides, the wakelocks it created lasted for about 1 minute on average. Obviously, the wakelocks were not released on time. Google Play service is used to update applications downloaded from Google Play Store, and synchronize application information. This application also applies for wakelocks too frequently. Normally, we only need to synchronize application information with the server a few times a day.

Other applications, such as *Google News & Weather*, *Skype*, *Books* and *Calendar*, even though generate fewer wakelocks, their wakelocks last too long. Obviously, Skype always

blocks the device to sleep when it is running in the background. It has a non-sleeping bug. The other three applications request wakelocks to synchronize with servers; however, the sync should not last for several minutes. Wakelocks should be released as soon as the sync task finishes.

From the previous analysis, the wakelock usage of many applications violates the original design rule of the opportunistic suspending technique: wakelock is used to protect important tasks. From Figure 7.2, we can see that non-system applications cause more than half of the device-blocked time. Wakelock is both misused and abused by these applications. That's why we propose *WakeFilter* to filter unnecessary wakelock requests.

Table 7.2: The number of wakelocks requested by applications and the average wakelock length. The data is based on the device usage trace of user 6.

Application	Wakelock Numbers	Average Wakelock Length (s)
android	9214	37.25
WeChat	3558	60.21
Google Play Service	7320	18.11
Google News & Weather	115	262.14
Skype	2	2797.23
Books	25	169.17
Calendar	24	154.32
Facebook	203	10.82
Gmail	106	8.86
QQ Sports	499	1.61
Download Provider	3	94.55
systemui	3	108.40
rild	744	0.15
Deskclock	6	0.19
Others	402	3.24

7.4 System Design

We use the energy adaptation technique to design *WakeFilter*. It uses battery level as the trigger. We designed three adaptation levels (from 3 to 1) in *WakeFilter*. When the battery level decreases to a predefined threshold, the adaptation level will decrease by 1. As the value of adaptation level decreases, fewer wakelocks can block the device to sleep. When the device

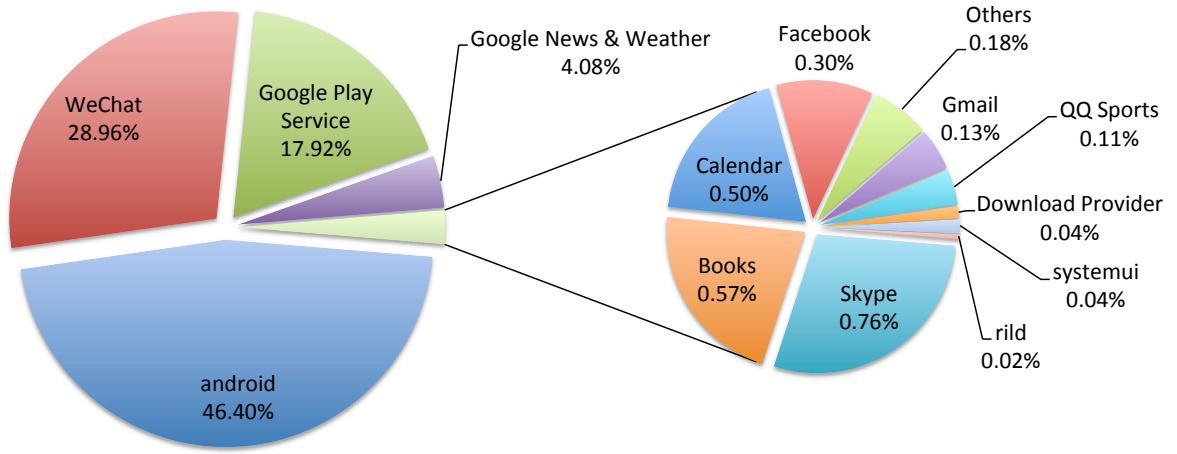


Figure 7.2: The percentage of time that the device was blocked by each application during the data collecting of user 6.

is connected to a power supply, the adaptation level is always set to 3. Accordingly, we also allocate a priority to wakelocks. The value of priority ranges from 1 to 3. Table 7.3 shows the relationship between adaptation level and wakelock priority. Whenever a wakelock, whose priority is less or equal to the adaptation level, is held, the system will be blocked to sleep.

Table 7.3: The relationship between adaptation level and wakelock priority.

Adaptation level	Wakelocks that can block the device to sleep
3	all wakelocks
2	wakelock priority ≤ 2
1	wakelock priority ≤ 1

7.4.1 Wakelock Categorization

Before designing the *WakeFilter* strategy, we analyzed the wakelocks in Android. Based on the creator of wakelock, we categorized wakelocks into three groups: application wakelock, system service wakelock, and kernel wakelock. The wakelocks of each group are generated by the creators in the corresponding layer of the system. For example, kernel wakelocks are generated by kernel modules, such as hardware drivers that need to keep the system active to handle important hardware events.

We reviewed the source code of Android and found out each place that creates wakelock.

For these system libraries and system applications, we assigned a default priority to each of them, shown as Table 7.4. Third-party application generated wakelocks are set dynamically based on the usage pattern. We defined the following rules to decide wakelock priority.

- We set the priority of foreground application's wakelock to 1.
- For applications that have a pattern item in the current context, we set their wakelocks' priority based on the support of the pattern item and a threshold value TH_w (the default value is 50%). If the support of the pattern item is greater than TH_w , we set it to 1. Otherwise, we set it to 2.
- For all other cases, the priority is set to 3.

7.4.2 Assign Priority to Application Requested Wakelocks

Based on application information, system states and usage pattern, *WakeFilter* decides the corresponding priority of each wakelock. When the foreground application changes or when the usage pattern is updated, it resets the priority of wakelocks accordingly. Furthermore, it listens to the power-connected and power-disconnected event and the battery-level event, and updates the adaptation level of the system accordingly. Listing 7.2 and 7.3 are the pseudo code to set the priority of third-party application generated wakelocks and decide the current adaptation level.

WakeFilter will process the following kinds of wakelock requests at application framework layer:

1. **Application generated wakelock:** These wakelocks can be categorized into two categories: the wakelocks that keep the processor active (partial wakelock) and the wakelocks that control the power state of the screen and keyboard backlight. These wakelocks come from the code that was written by application developers, and is usually used to ensure the execution of a critical part of the program or to keep the screen or backlight active.

System Layer	Name	Functionality	Group
<i>Application</i>	Voice dialer	Used for voice dialing.	1
	Caller Cache	Cache caller information to database.	2
	Notification Player	Notify the user by playing sound when a communication event is received.	1
	SMS Transaction	Process sending SMS transaction.	1
	Exchanger Service	Check and receive emails.	2
<i>Framework</i>	Power Command	Control the power manager to stay on when plugged in.	1
	Headset Base	Handfree device communication.	1
	Async Player	Library for playing audio.	2
	Location Manager	Manage location providers and issue location updates and alerts.	2
	CDMA Connection	APIs for CDMA connection.	1
	GSM Connection	APIs for GSM connection.	1
	RIL Service	RIL command interface.	1
<i>System</i>	Audio Hardware	Audio hardware driver.	1
	Audio Policy Service	Control the activity and configuration of audio input and output streams.	1
	Input Event Hub	Process and dispatch input events.	1
	GPS Location Provider	Location with satellites. Manage application request conditions.	2
	WLAN Loader	Wireless network loader.	1

Table 7.4: The wakelock creators in the Android system.

2. **Application library generated wakelock:** Most of these wakelocks are partial wakelocks and are used to protect the key part of the library code, such as the operation of writing caller information into the database. Since system developers wrote libraries, and higher-level APIs to encapsulate the wakelock operations, this kind of wakelocks rarely cause problems.
3. **System service generated wakelock:** System services are used to execute low-level operations, such as locating management. Similar to application libraries, most wakelocks generated by them are partial wakelocks, except for the power command service, which request a wakelock to keep the screen bright when the device is charging.

Listing 7.2: The pseudo code for deciding wakelock priority.

```

void setWakelockPriority(Wakelock w){
    if(w.app == foreground app){
        w.priority = 1;
    }
    else if(w.app has a pattern p) {
        w.priority = (p.support > thW) ? 1 : 2;
    }else{
        w.priority = 3;
    }
}

```

Listing 7.3: The pseudo code for deciding adaptation level.

```

int getAdaptationLevel(){
    if(device is charging or battery level > thB1){
        return 3;
    }
    return (battery level > thB2) ? 2 : 1;
}

```

7.4.3 Decide When the Device Goes to Sleep

All the kernel managed wakelocks are partial wakelocks, which ensure the processor stays active. Some of the wakelocks have an expiration time. These wakelocks will be inactive when they expire regardless whether they were directly released or not. The system will suspend to memory only when there are no active wakelocks that have no expiration time, and all the wakelocks with an expiration time are expired. The objects that directly request wakelocks from kernel include Bluetooth driver, NFC driver, audio service, location service, EventHub,

sensor service, ril service, system and power manager service.

Different to the original design of Android, the new design does not consider wakelocks that have a higher priority than the system's adaptation level. The pseudo code of suspend-checking is shown as Listing 7.4. From the pseudo code, we can also see that a timer will be added to trigger the system to check suspend state again if some wakelocks have an expiration time. Otherwise, the power management system will execute the suspend-to-memory procedure and prepare to sleep. The power management system invokes this procedure when the power management system receives a wakelock request or release event. After we update the priority of wakelocks, it also rechecks the suspending state of the system and suspends the system if all conditions are satisfied.

Listing 7.4: The pseudo code of the suspend-checking procedure.

```
bool suspend_checker(){
    int max_timeout = 0;
    foreach(wakelock w in active queue){
        if(w.privilege > adaptation state)
            continue;
        else if(w.has_expiration_time){
            if(w.timeout <= 0)
                expire_wakelock(w);
            else if(w.timeout > max_timeout)
                max_timeout = w.timeout;
        } else {
            return false;
        }
    }
    update_timer(max_timeout);
}
```

```

    return max_timeout == 0;
}

void update_timer(max_timeout){
    close interrupts;
    if(max_timeout > 0)
        modify_expire_timer(max_timeout);
    else{
        delete_expire_timer();
        suspend_to_memory();
    }
    resume interrupts;
}

```

7.5 Implementation

We implemented the *WakeFilter* mechanism in Android 4.4.2 and Linux kernel 3.4.0. We modified the kernel layer, the native library layer and the application framework layer of Android. In this section, we describe the implementation in detail.

7.5.1 Energy Adaptation Support

In Linux kernel, we added the “*adapt_level*” system attribute to represent the current adaptation level of the system, and the corresponding system file “*/sys/power/adapt_level*” as the interface between kernel space and user space. When we write or read a *sysfs* file from user space, the related “*store*” function or “*show*” function of this system attribute will be invoked. For example, when we write an integer value to the “*/sys/power/adapt_level*” file, the “*adapt_level_store*” function will read this value and update the kernel. In the native library layer, we added the “*set_adapt_level*” function to the legacy power library, it writes the new adaptation level to the “*/sys/power/adapt_level*” system file. In the application framework

layer, we defined the “*setAdaptLevel*” and “*getAdaptLevel*” APIs in the “*IUPSPowerManager*” AIDL interface, and implemented them in the “*UPSPowerManager*” and “*UPSPowerManagerService*” classes, as well as its corresponding JNI (Java Native Interface) layer. When a system service or an application invokes the “*setAdaptLevel*” function, the operation, follows the implementation in these layers, writes the value to the system file. The “*setAdaptLevel*” method not only spreads the new adaptation state to the kernel through the kernel layer interface, but also broadcast an “*Intent*” message to notify all the energy-aware mechanisms.

To support energy adaptation, we added a new strategy to the *UPS* system named “*EnergyAdaptationStrategy*”, and implemented the strategy we described in Listing 7.3 in the “*update*” method to decide the new adaptation state and invokes the “*setAdaptLevel*” method of “*UPSPowerManager*” to update the adaptation level of system. It listens to update event of charging state and screen state, and the *UPS* system invokes its “*update*” method when one of them is received. This strategy increases the adaptation level when the battery level reaches a predefined threshold and resets the adaptation level to 3 when the device is charging.

7.5.2 Prioritized Wakelock Support

To add priority to wakelocks, we changed all the layers of Android. In Linux kernel, we first added the priority field to the wakelock object. Then we changed the API of the system file “*/sys/power/wake_lock*” to take priority as a parameter. We added wakelock priority to wakelock name in the format “*name|priority*”, and parsed the wakelock priority in the “*lookup_wake_lock_name*” function of the power manager module in Linux kernel. In application framework layer, we implemented the strategy to decide wakelock priority, as described in Listing 7.2, in the “*UPSPowerManagementService*” class, which receives all the wakelock operations. Moreover, it listens to the update event of user behavior, which is sent from the *UPS* system, and invokes the “*acquire_wake_lock*” and “*release_wake_lock*” functions of the legacy power library to communicate with the kernel. Updating the priority of a wakelock is simply a combination of release and request operations.

7.5.3 Wakelock Filtering

Wakelock filtering is implemented in the kernel layer and the application framework layer. In the application framework layer, all the wakelock requests of the application layer are combined to a single wakelock (we name it as **AWL**). This wakelock will further be transmitted to the kernel with other wakelock requests generated by library layer creators. In this way, the kernel only manages a very limited amount of wakelocks. So that, at the application framework layer, we update the state of **AWL** based on the states of all the application wakelocks and request/release it accordingly. Differing from the original design, we need to add the adaptation level to the **AWL** wakelock and update it whenever an application applies or releases a wakelock. Finally, we hold it or update its level in the “*updateSuspendBlockerLocked*” function through the kernel layer API of wakelock.

For the kernel layer, we modified the “*has_wake_lock_locked*” function to decide the suspending state by considering the adaptation level system and wakelock. In this way, when the system enters into the energy adaptation mode, the system gets more chances to suspend to memory and save a dramatic amount of battery energy. We also added another function called “*update_timer*”, which will be invoked when the adaptation level changes. This function invokes the “*has_wake_lock_locked*” function to get the correct expiration time after the system state changes. Then it updates the timer if the expiration time changes or adds the suspend-to-memory task to work queue (start to suspend) if no wakelocks with a lower-priority are held.

In the library layer, we also changed the *JNI (Java Native Interface)* of the “*UPSPowerManagementService*” and the legacy power hardware layer of Android. They write the power management commands to the kernel space. In this way, we build a hierarchical system across the whole four layers of Android.

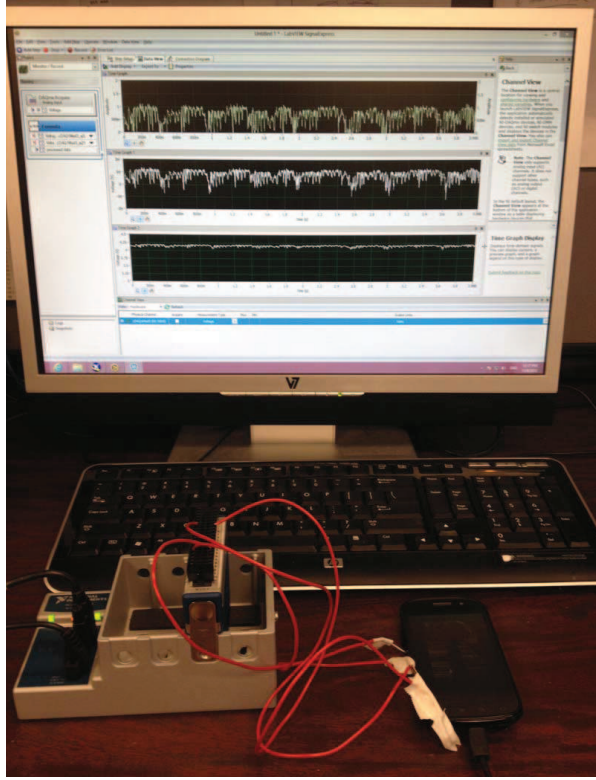


Figure 7.3: The power measuring platform with NI cDAQ-9174.

7.6 Evaluation

We used two methods to evaluate the effectiveness of the *WakeFilter* strategy. First, we evaluated the energy consumption of four specific usage cases. Second, we simulated the power consumption of two usage scenarios based on the device usage traces of 14 users.

7.6.1 Method

First of all, we evaluated the power consumption in the device idle state. For all the scenarios, we look at the first 5 minutes after display timeouts. This test includes four scenarios:

Scenario A WakeFilter is enabled. Adaptation level is 3. No third-party applications are installed on the device.

Scenario B WakeFilter is enabled. Adaptation level is 1. No third-party applications are installed on the device.

Scenario C WakeFilter is enabled. Adaptation level is 3. A group of third-party applications, such as Facebook and Youtube, are installed on the device.

Scenario D WakeFilter is enabled. Adaptation level is 1. The same group of third-party applications as Scenario C are installed on the device.

In the experiment of these four scenarios, we used NI cDAQ-9174, which samples 100 times per second, to measure the power consumption of the device. We connected a very small shunt resistor, about 0.005Ω , into the position electrode of the cell phone, and then used the voltage monitor to measure the voltage of the resistor and the system. Finally, we used the inputs as parameters to compute the power consumption of the whole device. Figure 7.3 shows our experiment platform.

Second, we used simulation, the same as that was used in Chapter 5, to evaluate the effective of *WakeFilter* based on the usage traces. In the experiment, we compared how long the device worked in the sleep state and used the average idle power and active power of each user to estimate the extended battery life. For the following two scenarios, each user installed the applications they used on their personal device to the experiment device. During the experiment period, the users used the experiment devices as on their own devices.

Scenario E WakeFilter is enabled. Adaptation level is 3. Use the device usage traces of 14 users as inputs.

Scenario F WakeFilter is enabled. Adaptation level is 1. Use the device usage traces of 14 users as inputs.

7.6.2 Experiment Results Analysis

The *WakeFilter* strategy saves energy when the device is in the idle state. When the device is active, our mechanism does not save more energy, because the system holds a full wakelock to make the device active. In this section, we mainly discuss the power-saving when the system is idle.

Idle Usage Case Evaluation

First, we did two experiments to evaluate wakelock usage in scenario A and B. All the activities were generated by system services and build-in applications. We set the adaptation level to different values and run at each level for about 5 minutes. Figure 7.4 and Figure 7.5 are the monitored power of these two experiments. We can see that in Figure 7.5 there are more periods that the system works in the low-power mode. When the adaptation level is 3, the average power of the device was 1.14 watts; when the adaptation level is 1, the average power of the device was 0.997 watts. The energy saving is about 12.54%. From Figure 7.5, we can see that there was still a large amount of time that the system is blocked by wakelocks, which means that there is still a large space for energy optimization, because most activities are “meaningless” to users.

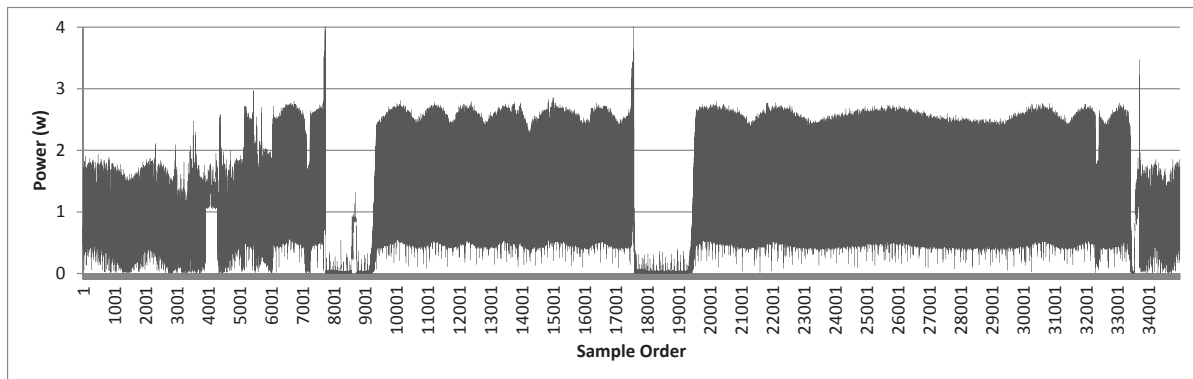


Figure 7.4: The measured power consumption of Nexus 4 after the display was automatically turned off. The adaptation level of the system is 3. No third-party applications were installed on the device.

In the second experiment, we started a group of applications (including *Facebook*, *Twitter*, *CNN*, *Gmail*, *Dropbox*, *AccuWeather* and *Amazon Mobile*), most of which will generate background activities. We performed the similar process as the previous experiment. The power consumption results of Scenario C and D are shown as Figure 7.6 and Figure 7.7 respectively. First, we can see that at the beginning of both Figures, the power dissipation is high because after the system sleeps, there were still some tasks to execute. Then, in Figure 7.7 the system

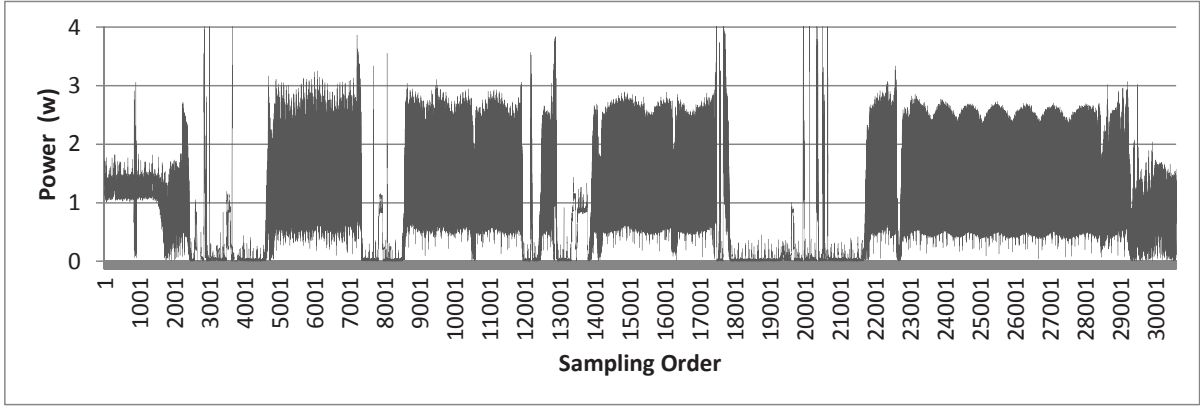


Figure 7.5: The measured power consumption of Nexus 4 after the display was automatically turned off. The adaptation level of the system is 1. No third-party applications were installed on the device.

suspended earlier than in Figure 7.6. Furthermore, in Scenario D, the result shows more low-power period than in Scenario C. The average power dissipation is 1.02 watts in Scenario D, and the average power is 1.27 watts in Scenario C. The energy saving is about 18.89%.

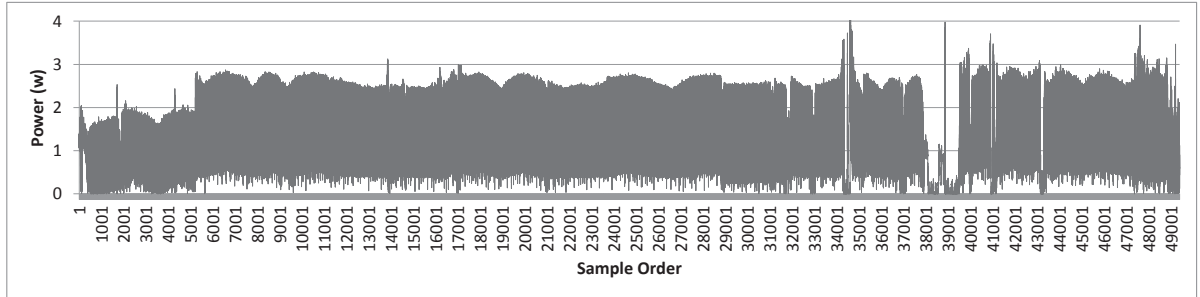


Figure 7.6: The measured power consumption of Nexus 4 after the display was automatically turned off. The adaptation level of the system is 3. A group of third-party applications were installed on the device and ran in the background.

Simulation Result Analysis

We implemented the wakelock mechanism in the simulator and analyzed how long wakelocks block the device to sleep. We enabled *WakeFilter* in the experiment but set the adaptation level to 3 and 1 separately. When the adaptation level is 3, all wakelocks can block the device to sleep. It is the same as the situation without *WakeFilter*.

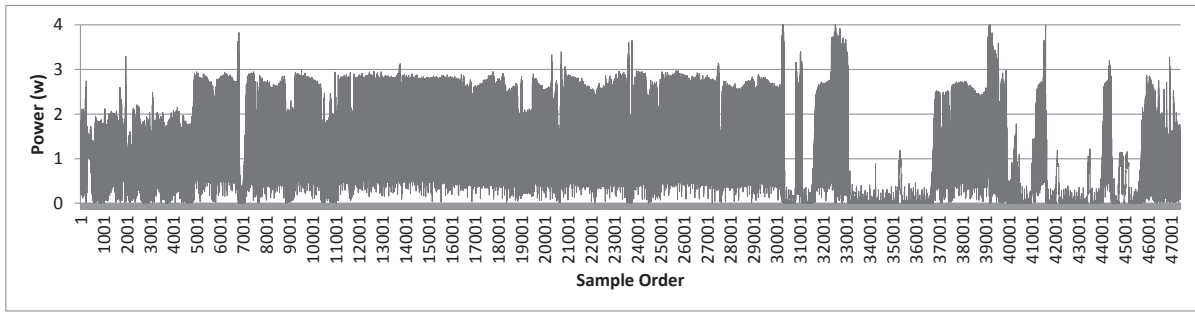


Figure 7.7: The measured power consumption of Nexus 4 after the display was automatically turned off. The adaptation level of the system is 1. A group of third-party applications were installed on the device and ran in the background.

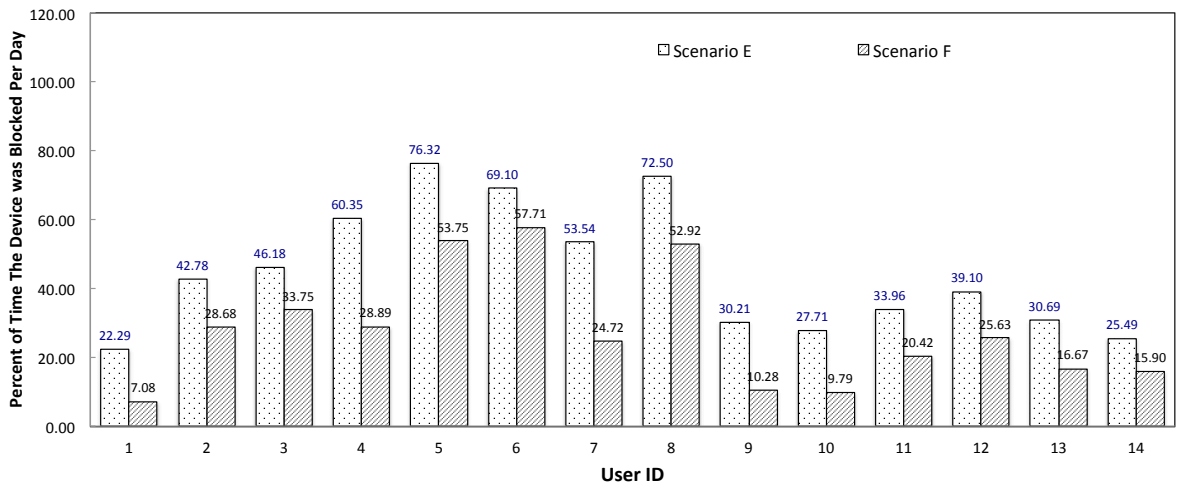


Figure 7.8: Percent of time the devices were blocked by wakelocks per day when WakeFilter was enable and the adaptation level was 3 and 1.

In the analysis, we did not count the wakelocks in two situations: device charging and device active. When the device is charging, WakeFilter will be disabled. Wakelocks will always block the device to sleep when the device is active, it is meaningless to analyze the wakelocks in this situation. If a wakelock crosses the device active and idle state, we only count the part of the time in the idle state. We analyze the power saving in the best case, and ignored the power used to execute the suspended background applications (when the device becomes active, they will be executed together).

Compared to Scenario E, we found the device-blocked time (the time that the device was blocked to sleep by wakelocks) was significantly decreased in Scenario F, as shown of Fig-

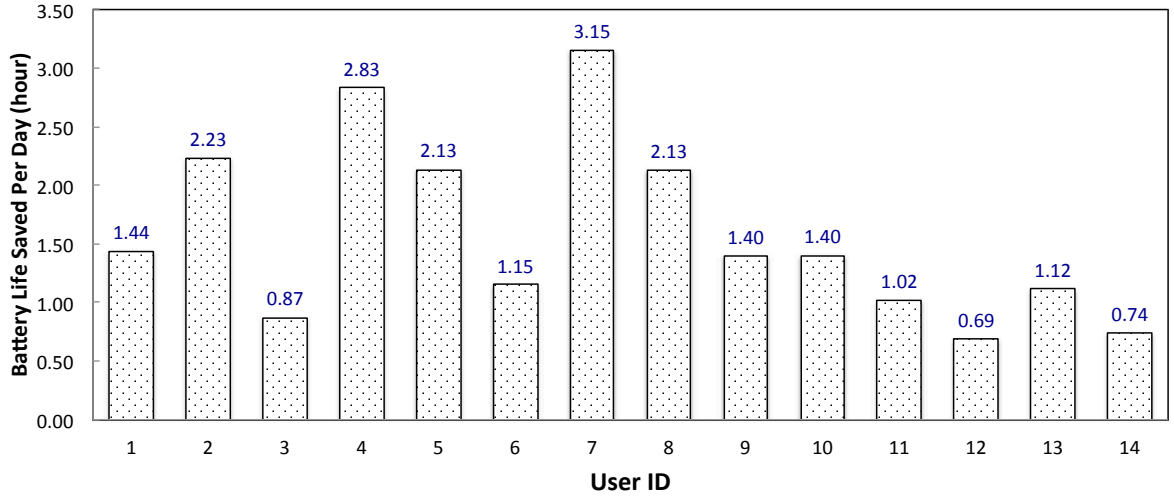


Figure 7.9: Battery life saved per day when WakeFilter was enabled and the adaptation level is 1.

ure 7.8. On average, only 27.58% of time the devices were blocked in Scenario F, and as much as 45.01% of time the devices were blocked by wakelocks in Scenario E. This shows that WakeFilter can effectively eliminate the unnecessary wakelocks and make the device sleep much longer. Because WakeFilter can only make the device sleep when the device is idle, we use the average device idle power of each user, as shown of Table 5.1, to calculate the saved energy. We use the average device active power to calculate the extended battery life. From Figure 7.9, we can see that in Scenario F, battery life can be extended for about 1.58 hours per day on average. For some of the users, it can be extended for more than 2 hours.

7.7 Related Work

In this section, we discuss the previous publications about optimizing the wakelock mechanism. Pathak *et al.* [36] found the “no-sleep bug” for the first time. Vekris *et al.* presented a tool to detect “no-sleep bugs” in [37]. They used an inter-procedural data flow analysis framework to verify if an application’s wakelock usage has this problem. Jindal *et al.* found an energy bug named sleep conflict in [100]. It may happen in device drivers when switching from the high-power state to the low-power state.

To solve the misuse of wakelocks, Kim *et al.* [39] proposed PR-wakelock to predict the

misuse of wakelocks and forcibly release these wakelocks. In their new wakelock mechanism, the function call analysis was used to analyze the behavior of wakelocks, and to predict possible of misuse. The improper wakelock will be automatically released to make the system get more chances to sleep. Similarly, a misused case of wakelocks was detected in [38]. They proposed *WakeScope* to monitor wakelock behavior via probing kernel functions. Then, they detected if a wakelock is misused when the application state changes. In [40], Alam *et al.* proposed a data flow based analysis strategy to determine the placement of wakelock statements.

Different with these previous works, we solve the problem with energy adaptation and user-centric system design. We use the usage patten of users to find out whether the task of each application is important. If the application has a pattern at the given context, we will set the priority of this application generated wakelocks to a high priority. Otherwise, the priority of wakelocks is low, and the wakelock will not block the device to sleep in the device idle state.

7.8 Summary

In this chapter, we analyzed the wakelock usage based on the usage traces of 14 users. Based on our analysis, we found wakelocks are commonly misused and abused in Android. On average, about 50.83% of the idle period of the device was blocked by wakelocks. They make the device cannot work in the low-power state, and consume a considerable amount of battery energy.

To solve the problem, we propose the *WakeFilter* strategy to filter low-priority wakelocks in the energy adaptation state. Such that, these wakelocks generated by applications will not block the device to sleep in the idle state. We use the usage pattern of the individual user to decide the priority of application generated wakelocks. So that important application's tasks will be guaranteed to execute, and unimportant application's tasks are suspended to execute.

We used two methods to evaluate the energy consumption in the special usage scenario and normal device usage scenario. The experiment result shows that *WakeFilter* can save about 12.54% and 18.89% battery energy consumption in two special usage scenarios. Moreover,

the simulation result found only about 27.58% idle period is blocked by wakelocks when using *WakeFilter*, as corresponded to 45.01% in the normal case. Based on the average idle power and active power from users' traces, we found *WakeFilter* can extend battery life by as much as 1.58 hours per day on average for these users.

CHAPTER 8

CONCLUSION

Power management is becoming more and more important when designing mobile systems. Traditional power management systems use unified power management policies to save battery energy through controlling the power state of hardware components. However, recent researches found this method cannot work effectively to optimize the power consumption of the system because applications activities make hardware components frequently work in high-power mode. Most of the applications are redundant to the user. New generation power-saving techniques were also proposed to solve the power problem, but they usually failed to consider the impact to user experience. In this dissertation, we proposed the user-centric power management to dynamically customize the power-optimization strategies based on the user behavior of individual user. From the analysis of 14 users' device usage traces, we found the user behavior of most users follows a pattern, which helps us to distinguish important applications at different context. Thus, we can use usage pattern to restrict the power draw of redundant application activities. In this chapter, we conclude the work we have done in this dissertation.

8.1 Conclusion

The most important and difficult step for user behavior-related research is data collecting. We modified the Android system to collect user behaviors and the corresponding system events. We selected 14 volunteers as the experiment target and collected their usage traces with the experiment platform we supplied. We spent about eight months to finish the two-stage data collecting work and collected about 7GB data from users. Then we implemented data analysis programs to analyze the charging pattern of users, the application usage pattern, battery power consumption in different states, application power consumption, location service usage, and wakelock usage. From this analysis, we found the battery life of our experiment device is very poor. Users charged the devices more than one times a day, and the charged power can only support the device to be active for 11.89% of the time during a whole day on average.

A significant amount of battery energy was consumed by background application activities, which are either unnecessary or unneeded to the user.

We also designed and implemented statistical and data mining algorithms to analyze the usage pattern of users. We used these algorithms to analyze the interest points, active periods, and usage pattern. From the analysis result, we found user behavior is both time-dependent and location-dependent. Nearly all users have their personal interest points and active periods. In these contexts, the users are also inclined to use a specific group of applications, which are important to the users. The association rules between context and application are valuable for user-centric system design. We can utilize it to distinguish important application activities from normal application activities, and then we can take actions to restrict or reschedule the redundant application activities to save battery energy. Besides, these rules are also helpful for other aspects of user-centric design, such as improving the UI design of applications.

Based on the observations, we designed and implemented the *UPS* system to bridge user behavior and energy-saving strategies. In the *UPS* system, we narrow down user behavior to user-app interactions. The *UPS* system collects user-app interaction events to analyze the usage pattern of users. As a assistant framework for designing user-centric energy-saving strategies, it also listens to key system events and interval events to trigger the registered power-optimization operations defined in the strategies. With this system, researches can concentrate on designing power optimization policies. Besides, we designed the *UCASS* strategy to eliminate the power consumed by redundant application activities. It uses the usage pattern of users to find out unimportant application. Then, it reschedules the background tasks of these applications to the following device charging period or the next device active period, such that we can reduce their power consumption with task grouping. Our experiment result shows the average saved battery energy consumed by background application activities is about 25.62%.

In addition, we found most location requests were duplicated because users usually used mobile device very heavily in several dedicated locations. We proposed *LocalLite* to remove

the energy waste caused by duplicated requests. The *LocalLite* location provider combines location with wireless access points, which are mostly immovable, and caches received locations. Such that the location requests generated in the same context do not have to power on hardware components to pinpoint user location. This improvement can significantly reduce the redundant location requests while at the same time returns accurate location information. On average, *WakeFilter* saves about 98.51 percent of energy consumed by location requests. It also enables location-marked user behavior sensing.

Finally, we proposed the *WakeFilter* strategy to optimize the opportunistic suspending mechanism of Android because we found wakelock misuse and abuse are common in Android. Many applications aggressively use wakelocks and some of them even fail to release wakelocks. The *WakeFilter* strategy utilizes user's usage pattern to decide the priority of applications' wakelocks. When the system works in the energy adaptation mode, low-prioritized wakelocks cannot block the device to sleep. *WakeFilter* decreases the time that the device was blocked to sleep from 45.01% to 27.58% when the device is idle. The average extended battery life is about 1.58 hours per day.

CHAPTER 9

FUTURE WORK

In the future, we can improve the *UPS* system several aspects. In this section, we discuss the future work of this dissertation.

9.1 Future Work

The power management of mobile devices is a tough task as energy-waste exists in different aspects of the system. There is nearly no way to solve the problem once for all. Besides, with the improvement of hardware performance and the advancement of application design, new issues that cause battery energy waste emerge every year. In the future, we still need to do a lot of work to improve battery life.

First, more work should be done on the research of user behavior. We need to monitor user behavior in a much longer period, so that we can analyze how frequently user behavior changes with time. With this result, we can set a more accurate length to the sensing period of the *UPS* system. Also, we need to expand the idea of user behavior to other aspects and collect the corresponding data from real users. Based on the new style of usage pattern, we can observe more energy waste in the system.

Second, more energy-saving strategies should be developed based on the usage pattern supplied by the *UPS* system. We believe that task grouping is not the only way to use the usage pattern, it can also be used to design other energy-saving strategies to diminish the battery energy wasted by applications. The usage pattern can also be used to optimize the traditional power management systems, such as dynamically setup the power-related configurations of hardware components. For example, we can dynamically set the timeout time of display based on active period. During the active period, the user uses the device more frequently. We can set it to a larger value based on the interaction intervals. In other time periods, we can set it to a smaller value. In this way, the energy consumption of display can be saved without losing too much user experience.

Third, other system services' usage should as well be investigated to find out the redundant usage, and use the usage pattern to optimize it. We noticed that the *android* application consumes a large amount CPU time when the device is either active or idle. Thus, there is a great potential to save energy through optimizing system services. The relationship between applications and system services should also be investigated, such that we know how services generates activities while applications are using them. The result is also helpful for developers to design energy-aware applications.

Finally, the simulator we used should also be optimized to estimate power consumption of applications in different situations. We should find out how the power of system services change with applications. Besides, we should expand the simulator to estimate the power consumption for more platforms and more system configurations. In this dissertation, we calibrate the power consumption of applications, specifically for the experiment device. Afterward, the power models should be able to self-calibrating. Such that, we can compare the effectiveness of the energy-saving models for different devices.

APPENDIX

A.1 User Behavior Analysis Algorithms

A.1.1 Interest Point Analysis Algorithm

Listing A.1: The pseudo code of the interest point analysis algorithm.

```

ArrayList<InterestPoint>
analyzeInterestPoints ( ArrayList<AppUsage> aus ){
    Map<Integer , InterestPoint> map =
        new HashMap<Integer , InterestPoint >();
    int interactions = 0;
    for(AppUsage au : aus){
        InterestPoint ip = map.get(au.getLocationHash());
        if(ip == null){
            ip = new InterestPoint(au.location);
            map.put(au.getLocationHash() , ip);
        }
        ip.addInteraction();
        interactions++;
    }

    List<InterestPoint> points=
        new ArrayList<InterestPoint >();
    points.addAll(map.values());
    double maxDensity = 0;
    for(InterestPoint point : points){
        point.computeDensity(interactions);
    }
}

```

```

        if (point.getDensity() > maxDensity)
            maxDensity = point.getDensity();
    }

    double avgDensity = 100.0/points.size();
    double coreDensity = avgDensity
        + (maxDensity-avgDensity)*ipStandard/100.0;
    for(int i = points.size() - 1; i >= 0; i--){
        if (points.get(i).getDensity() < coreDensity)
            points.remove(i);
    }
    return points;
}

```

A.1.2 Active Period Analysis Algorithm

Listing A.2: The code of the active period analysis algorithm.

```

ArrayList<TimeInterval> analyzeTimeIntervals(){
    ArrayList<TimeInterval> result =
        new ArrayList<TimeInterval>();
    int count = timeUnit * 24 * 6;
    for(int i = 0; i < count; i++){
        result.add(
            new TimeInterval(i*10, (i+1)*10, timeUnit));
    }

    int inumber = 0;
    for (Map.Entry<Integer, App> entry : map.entrySet()) {

```

```

App app = entry.getValue();
for(AppUsage au : app.getAppUsages()){
    inumber += markPoints(au, result);
    if(au.endTime > executeTime)
        executeTime = au.endTime;
    if(au.beginTime < statStartTime)
        statStartTime = au.beginTime;
}
}

double[] standard = tpStandard[timeUnit == 7 ? 1 : 0];
double max = 0, density;
for(TimeInterval ti : result){
    density = ti.calculateDensity(inumber);
    if(density > max) max = density;
}

density = standard[1]
    +(max-standard[1])*standard[0]/100.0;
for(TimeInterval ti : result){
    ti.setPointType(inumber, density, standard[1]);
}

TimeInterval ti, left, right;
int index;
for(int i = 0; i < result.size(); i++){

```

```

ti = result.get(i);
switch( ti.getType() ) {
    case CORE:
        cluster ti.setCluster(i);
        break;
    case BORDER:
        for(int j = 1; j <= tpDistance; j++){
            index = (i - j + result.size()) % result.size();
            left = result.get(index);
            if( left.getType() == PointType.CORE){
                ti.setCluster(index);
                break;
            }

            index = (i + j) % result.size();
            right = result.get(index);
            if( right.getType() == PointType.CORE){
                ti.setCluster(index);
                break;
            }
        }
        break;
    default:
        break;
}
}

```

```

ArrayList<TimeInterval> intervals =
    new ArrayList<TimeInterval>();
TimeInterval last = null;
int clusterSN = 0;
for(TimeInterval ti2 : result){
    if(ti2.getCluster() >= 0){
        if(last == null){
            last =
                new TimeInterval(ti2.start, ti2.end, timeUnit);
            last.setCluster(clusterSN++);
        }else{
            last.end = ti2.end;
        }
    }else{
        if(last != null){
            intervals.add(last);
            last = null;
        }
    }
}

```

```

if(last != null) { intervals.add(last); }
if(intervals.size() == 0) return intervals;
last = intervals.get(intervals.size() - 1);
for(int i = intervals.size() - 2; i >= 0; i--){

```



```

        ti = intervals.get(i);
        if(last.start - ti.end <= 30){
            last.start = ti.start;//combine
            intervals.remove(i);
        }else{
            last = ti;
        }
    }
    return intervals;
}

int
markPoints(AppUsage au, ArrayList<TimeInterval> times){
    Calendar start = Calendar.getInstance();
    start.setTimeInMillis(au.beginTime);
    int minutesBetween =
        (int)((au.endTime-au.beginTime)/1000/60);
    if(minutesBetween <= 0)
        return 0;

    int hour = start.get(Calendar.HOUR_OF_DAY);
    int minute = start.get(Calendar.MINUTE);
    int i = hour * 6 + minute/10;

    if(timeUnit == 7){
        int date = start.get(Calendar.DAY_OF_WEEK);

```

```

        i += (date - 1) * 24 * 6;
    }

    int result = 0;
    for(int t=0; t < minutesBetween;
        i=(++i)%times.size(), t+=10){
        times.get(i).increaseInteraction();
        result++;
    }

    return result;
}

```

A.1.3 Usage Pattern Analysis Algorithm

Listing A.3: The code of the usage pattern analysis algorithm.

```

ArrayList<UsagePattern>
analyzeUsagePattern(ArrayList<TimeSlice> slices ,
    ArrayList<InterestPoint> points){
    HashMap<Integer , UsagePattern> patternMap =
        new HashMap<Integer , UsagePattern>();

    for(Map.Entry<Integer , App> entry : map.entrySet()){
        App app = entry.getValue();
        for(AppUsage au : app.getAppUsages()){
            InterestPoint ip = findInterestPoint(au, points);
            if(ip == null) { continue; }
            ArrayList<TimeSlice> tis = findInterval(au, slices);

```

```

    for(TimeSlice ti : tis){
        int hash = UsagePattern.hashCode(ti , ip , app);
        UsagePattern p = patternMap.get(hash);
        if (p == null) {
            p = new UsagePattern(ti , ip , app);
            patternMap.put(hash , p);
        }
        p.updateUsage(au , executeTime);
    }
}

int statSpan =
    (int)Math.ceil(((executeTime-startTime)/3600000/24))+1;
ArrayList<UsagePattern> patterns =
    new ArrayList<UsagePattern>();

for(Map.Entry<Integer , UsagePattern> entry :
        patternMap.entrySet()){
    UsagePattern pattern = entry.getValue();
    if(pattern.getSupport(statSpan) > minSupport)
        patterns.add(pattern);
}

return patterns;
}

```

```

ArrayList<TimeSlice>
findInterval(AppUsage au, ArrayList<TimeSlice> slices){
    ArrayList<TimeSlice> result = new ArrayList<TimeSlice>();
    int[] span = au.timeToMinutes(timeUnit);

    for(TimeSlice ti : slices){
        if(!(span[0] > ti.end || span[1] < ti.start))
            result.add(ti);
    }

    return result;
}

InterestPoint
findInterestPoint(AppUsage au,
                  ArrayList<InterestPoint> points){
    for(InterestPoint ip : points){
        if(ip.distanceBetween(au.location[0],
                              au.location[1]) < 10){
            au.iPoint = ip;
            return ip;
        }
    }
    return null;
}

```

REFERENCES

- [1] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *SIGOPS Oper. Syst. Rev.*, 34:13–14, April 2000.
- [2] David Snowdon, Sergio Ruocco, and Gernot Heiser. Power management and dynamic voltage scaling: Myths and facts, September 2005.
- [3] Aaron Carroll and Gernot Heiser. Mobile multicores: Use them or waste them. In *Proceedings of the Workshop on Power-Aware Computing and Systems, HotPower '13*, pages 12:1–12:5, New York, NY, USA, 2013. ACM.
- [4] Susanne Albers and Antonios Antoniadis. Race to idle: New algorithms for speed scaling with a sleep state. *ACM Trans. Algorithms*, 10(2):9:1–9:31, February 2014.
- [5] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Trans. Parallel Distrib. Syst.*, 21(5):658–671, 2010.
- [6] A. Rice and S. Hay. Decomposing power measurements for mobile devices. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 70 –78, 29 2010-april 2 2010.
- [7] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, February 2013.
- [8] Minyong Kim, Young Geun Kim, Sung Woo Chung, and Cheol Hong Kim. Measuring variance between smartphone energy consumption and battery life. *Computer*, 47(7):59–65, 2014.
- [9] Fred Douglass, P. Krishnan, and Brian N. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-*

- Independent Computing*, pages 121–137, Berkeley, CA, USA, 1995. USENIX Association.
- [10] Krisztián Flautner and Trevor Mudge. Vertigo: automatic performance-setting for linux. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 105–116, New York, NY, USA, 2002. ACM.
 - [11] Trevor Pering, Yuvraj Agarwal, Rajesh Gupta, and Roy Want. Coolspots: reducing the power consumption of wireless mobile devices with multiple radio interfaces. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, MobiSys '06, pages 220–232, New York, NY, USA, 2006. ACM.
 - [12] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor: past, present and future. In *Proceedings of Linux Symposium*, volume 2, pages 223–238, 2006.
 - [13] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. *SIGOPS Oper. Syst. Rev.*, 40(4):403–414, April 2006.
 - [14] Aaron Schulman, Vishnu Navda, Ramachandran Ramjee, Neil Spring, Pralhad Deshpande, Calvin Grunewald, Kamal Jain, and Venkata N. Padmanabhan. Bartendr: a practical approach to energy-aware cellular data scheduling. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, MobiCom '10, pages 85–96, New York, NY, USA, 2010. ACM.
 - [15] Venkatesh Pallipadi and Adam Belay. cpuidle - do nothing, efficiently... In *Proceedings of the Linux Symposium*, volume 2, June 2007.
 - [16] Suresh Siddha, Venkatesh Pallipadi, and Arjan Van De Ven. Getting maximum mileage out of tickless. In *Proceedings of the Linux Symposium*, pages 201–207, June 2007.
 - [17] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for os-level power management. In *Proceedings of the 4th ACM European*

- conference on Computer systems*, EuroSys '09, pages 289–302, New York, NY, USA, 2009. ACM.
- [18] David Snowdon. *Operating System Directed Power Management*. PhD thesis, University of New South Wales, March 2010.
- [19] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 16–16, Berkeley, CA, USA, 2011. USENIX Association.
- [20] Bhojan Anand, Karthik Thirugnanam, Jeena Sebastian, Pravein G. Kannan, Akhihebbal L. Ananda, Mun Choon Chan, and Rajesh Krishna Balan. Adaptive display power management for mobile games. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 57–70, New York, NY, USA, 2011. ACM.
- [21] Jie Liu, Bodhi Priyantha, Ted Hart, Heitor S. Ramos, Antonio A. F. Loureiro, and Qiang Wang. Energy efficient gps sensing with cloud offloading. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 85–98, New York, NY, USA, 2012. ACM.
- [22] Marcelo Martins and Rodrigo Fonseca. Application modes: A narrow interface for end-user power management in mobile devices. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, HotMobile '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.
- [23] Fengyuan Xu, Yunxin Liu, Thomas Moscibroda, Ranveer Chandra, Long Jin, Yongguang Zhang, and Qun Li. Optimizing background email sync on smartphones. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 55–68, New York, NY, USA, 2013. ACM.
- [24] R. Bolla, M. Giribaldi, R. Khan, and M. Repetto. Smart proxying: An optimal strategy

- for improving battery life of mobile devices. In *Green Computing Conference (IGCC), 2013 International*, pages 1–6, 2013.
- [25] Xiang Chen, Kent W. Nixon, Hucheng Zhou, Yunxin Liu, and Yiran Chen. Finger-shadow: An oled power optimization based on smartphone touch interactions. In *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*, Broomfield, CO, October 2014. USENIX Association.
- [26] Wook Song, Nosub Sung, Byung-Gon Chun, and Jihong Kim. Reducing energy consumption of smartphones using user-perceived response time analysis. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications, HotMobile '14*, pages 20:1–20:6, New York, NY, USA, 2014. ACM.
- [27] Sewook Park, Dongwon Kim, and Hojung Cha. Reducing energy consumption of alarm-induced wake-ups on android smartphones. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications, HotMobile '15*, pages 33–38, New York, NY, USA, 2015. ACM.
- [28] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mob. Netw. Appl.*, 18(1):129–140, February 2013.
- [29] Amin Vahdat, Alvin Lebeck, and Carla Schlatter Ellis. Every joule is precious: the case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, EW 9, pages 31–36, New York, NY, USA, 2000. ACM.
- [30] Andrew Grover. Modern system power management. *Queue*, 1(7):66–72, October 2003.
- [31] Manish Anand, Edmund B. Nightingale, and Jason Flinn. Ghosts in the machine: interfaces for better power management. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services, MobiSys '04*, pages 23–35, New York,

NY, USA, 2004. ACM.

- [32] Len Brown, Anil Keshavamurthy, David Shaohua Li, Robert Moore, Venkatesh Pallipadi, and Luming Yu. ACPI in Linux: Architecture, Advances and Challenges. In *Proceedings of the Linux Symposium*, volume 1, pages 51–67, Ottawa, Canada, July 2005.
- [33] Advanced Configuration and Power Interface Specification, July 2014. http://www.uefi.org/sites/default/files/resources/ACPI_5_1release.pdf.
- [34] Trinh Minh Tri Do, Jan Blom, and Daniel Gatica-Perez. Smartphone usage in the wild: A large-scale analysis of applications and context. In *Proceedings of the 13th International Conference on Multimodal Interfaces*, ICMI '11, pages 353–360, New York, NY, USA, 2011. ACM.
- [35] Choonsung Shin, Jin-Hyuk Hong, and Anind K. Dey. Understanding and prediction of mobile application usage for smart phones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 173–182, New York, NY, USA, 2012. ACM.
- [36] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.
- [37] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. Towards verifying android apps for the absence of no-sleep energy bugs. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.
- [38] Kwanghwan Kim and Hojung Cha. Wakescope: Runtime wakelock anomaly management scheme for android platform. In *Proceedings of the Eleventh ACM International*

- Conference on Embedded Software*, EMSOFT '13, pages 27:1–27:10, Piscataway, NJ, USA, 2013. IEEE Press.
- [39] Joonkyo Kim and Jaehyun Park. Reducing power consumption using improved wake-lock on android platform. In *The Eighth International Multi-Conference on Computing in the Global Information Technology*, ICCGI 2013, pages 171–174, Nice, France, 2013.
 - [40] F. Alam, P.R. Panda, N. Tripathi, N. Sharma, and S. Narayan. Energy optimization in android applications through wakelock placement. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–4, March 2014.
 - [41] Jonathan Corbet. A new approach to opportunistic suspend. <https://lwn.net/Articles/460644/>.
 - [42] Jonathan Corbet. Autosleep and wake locks. <https://lwn.net/Articles/479841/>.
 - [43] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132, 2002.
 - [44] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 139–152, New York, NY, USA, 2011. ACM.
 - [45] Salma Elmalaki, Mark Gottscho, Puneet Gupta, and Mani Srivastava. A case for battery charging-aware power management and deferrable task scheduling in smartphones. In *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*, Broomfield, CO, October 2014. USENIX Association.
 - [46] Jason Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Trans. Comput. Syst.*, 22(2):137–179, May 2004.
 - [47] Hui Chen, Bing Luo, and Weisong Shi. Anole: A case for energy-aware mobile applica-

- tion design. In *Proceedings of the 1st International Workshop on Power-Aware Systems and Applications*, Pittsburgh, PA, USA, September 2012.
- [48] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 225–238, New York, NY, USA, 2012. ACM.
- [49] Yifan Zhang, Xudong Wang, Xuanzhe Liu, Yunxin Liu, Li Zhuang, and Feng Zhao. Towards better cpu power management on multicore smartphones. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 11:1–11:5, New York, NY, USA, 2013. ACM.
- [50] Mian Dong and Lin Zhong. Power modeling and optimization for oled displays. *IEEE Transactions on Mobile Computing*, 11(9):1587–1599, September 2012.
- [51] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.
- [52] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.
- [53] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In *Proceedings of the 2012 USENIX Conference on Annual Tech-*

- nical Conference*, USENIX ATC'12, pages 36–36, Berkeley, CA, USA, 2012. USENIX Association.
- [54] Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. Devscope: A nonintrusive and online power analysis tool for smartphone hardware components. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 353–362, New York, NY, USA, 2012. ACM.
 - [55] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93, Washington, DC, USA, 2003. IEEE Computer Society.
 - [56] W. Lloyd Bircher and Lizy K. John. Complete system power estimation: A trickle-down approach based on performance events. *Performance Analysis of Systems and Software, IEEE International Symposium on*, 0:158–168, 2007.
 - [57] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM.
 - [58] Luca Benini and Giovanni de Micheli. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.
 - [59] J.R. Lorch and A.J. Smith. Software strategies for portable computer energy management. *Personal Communications, IEEE*, 5(3):60–73, Jun 1998.
 - [60] Srivatsa Vaddagiri, Anand K. Santhanam, Vijay Sukthankar, and Murali Iyer. Power management. *Linux J.*, 2004(119):11–, March 2004.
 - [61] Nilanjan Banerjee, Ahmad Rahmati, Mark D. Corner, Sami Rollins, and Lin Zhong.

- Users and batteries: Interactions and adaptive energy management in mobile systems. In *Proceedings of the 9th International Conference on Ubiquitous Computing, UbiComp '07*, pages 217–234, Berlin, Heidelberg, 2007. Springer-Verlag.
- [62] Ana Belen Lago and Iker Larizgoitia. An application-aware approach to efficient power management in mobile devices. In *Proceedings of the Fourth International ICST Conference on COMMunication System softWARE and middlewaRE, COMSWARE '09*, pages 11:1–11:10, New York, NY, USA, 2009. ACM.
- [63] ACPI. Advanced configuration power interface. <http://www.acpi.info/>.
- [64] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. Diversity in smartphone usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys '10*, pages 179–194, New York, NY, USA, 2010. ACM.
- [65] A. Rahmati and Lin Zhong. Studying smartphone usage: Lessons from a four-month field study. *Mobile Computing, IEEE Transactions on*, 12(7):1417–1427, July 2013.
- [66] Vijay Srinivasan, Saeed Moghaddam, Abhishek Mukherji, Kiran K. Rachuri, Chenren Xu, and Emmanuel Munguia Tapia. Mobileminer: Mining your frequent patterns on your phone. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '14*, pages 389–400, New York, NY, USA, 2014. ACM.
- [67] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '13*, pages 275–284, New York, NY, USA, 2013. ACM.
- [68] Suman Nath. Ace: Exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th International Conference on Mobile Systems, Ap-*

- plications, and Services*, MobiSys '12, pages 29–42, New York, NY, USA, 2012. ACM.
- [69] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, pages 47–56, New York, NY, USA, 2011. ACM.
- [70] ios 9 mobile operating system. <http://www.apple.com/ios/whats-new/>.
- [71] Gu. Chen, B. Kang, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and R. Chandramouli. Energy-aware compilation and execution in java-enabled mobile devices. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8 pp.–, April 2003.
- [72] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.
- [73] M. Segata, B. Bloessl, C. Sommer, and F. Dressler. Towards energy efficient smart phone applications: Energy models for offloading tasks into the cloud. In *Communications (ICC), 2014 IEEE International Conference on*, pages 2394–2399, June 2014.
- [74] Hao Qian and Daniel Andresen. Extending mobile device's battery life by offloading computation to cloud. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '15, pages 150–151, Piscataway, NJ, USA, 2015. IEEE Press.
- [75] Youhuizi Li, Hui Chen, and Weisong Shi. Power behavior analysis of mobile applications using bugu. *Sustainable Computing: Informatics and Systems*, 4:183–195, September 2014.

- [76] Hui Chen, Youhuizi Li, and Weisong Shi. Fine-grained power management using process-level profiling. *Sustainable Computing: Informatics and Systems*, 2:33–42, March 2012.
- [77] Frank Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, EW 9, pages 37–42, New York, NY, USA, 2000. ACM.
- [78] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.
- [79] Karan Aggarwal, Chenlei Zhang, Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia. The power of system call traces: Predicting the software energy consumption impact of changes. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, CASCON '14, pages 219–233, Riverton, NJ, USA, 2014. IBM Corp.
- [80] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 57–70, Berkeley, CA, USA, 2013. USENIX Association.
- [81] Kitae Kim, Donghwa Shin, Qing Xie, Yanzhi Wang, Massoud Pedram, and Naehyuck Chang. Fepma: Fine-grained event-driven power meter for android smartphones based on device driver layer event monitoring. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 367:1–367:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

- [82] Albert Ali Salah, Bruno Lepri, Alex Sandy Pentland, and John Canny. Understanding and changing behavior [guest editors' introduction]. *Pervasive Computing, IEEE*, 12(3):18–20, July 2013.
- [83] Mohammad Hossein Falaki. *Automating Personalized Battery Management on Smartphones*. PhD thesis, University of California at Los Angeles, Los Angeles, CA, USA, 2012. AAI3506477.
- [84] Martin Ester, Hans peter Kriegel, Jrg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2th International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996.
- [85] Albert Ali Salah, Bruno Lepri, Alex Sandy Pentland, and John Canny. Understanding and changing behavior [guest editors' introduction]. *Pervasive Computing, IEEE*, 12(3):18–20, July 2013.
- [86] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, MobiCom '15*, pages 40–52, New York, NY, USA, 2015. ACM.
- [87] R. Bolla, R. Khan, X. Parra, and M. Repetto. Improving smartphones battery life by reducing energy waste of background applications. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*, pages 123–130, Sept 2014.
- [88] Kaisen Lin, Aman Kansal, Dimitrios Lymberopoulos, and Feng Zhao. Energy-accuracy trade-off for continuous mobile device location. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys '10*, pages 285–

298, New York, NY, USA, 2010. ACM.

- [89] Kumar Vishal, Romil Bansal, Anoop M. Namboodiri, and C. V. Jawahar. Providing services on demand by user action modeling on smart phones. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, UbiComp '14 Adjunct, pages 167–170, New York, NY, USA, 2014. ACM.
- [90] A. Khairy, H.H. Ammar, and R. Bahgat. Smartphone energizer: Extending smartphone's battery life with smart offloading. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 329–336, July 2013.
- [91] The api of location service of android platform. <http://developer.android.com/guide/topics/location/strategies.html>.
- [92] The api of the location service of ios platform. <https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/LocationAwarenessPG/CoreLocation/CoreLocation.html>.
- [93] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
- [94] Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh. Improving energy efficiency of location sensing on smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 315–330, New York, NY, USA, 2010. ACM.
- [95] Yun Huang, A. Tomasic, Yufei An, C. Garrod, and A. Steinfeld. Energy efficient and accuracy aware (e2a2) location services via crowdsourcing. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, pages 436–443, Oct 2013.

- [96] He Wang, Zhiyang Wang, Guobin Shen, Fan Li, Song Han, and Feng Zhao. Wheel-loc: Enabling continuous location service on mobile phone for outdoor scenarios. In *INFOCOM, 2013 Proceedings IEEE*, pages 2733–2741, April 2013.
- [97] S.K. Datta, C. Bonnet, and N. Nikaein. Android power management: Current and future trends. In *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*, pages 48–53, June 2012.
- [98] Xigui Wang, Xianfeng Li, and Wen Wen. Wlcleaner: Reducing energy waste caused by wakelock bugs at runtime. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*, pages 429–434, Aug 2014.
- [99] P.S. Patil, J. Doshi, and D. Ambawade. Reducing power consumption of smart device by proper management of wakelocks. In *Advance Computing Conference (IACC), 2015 IEEE International*, pages 883–887, June 2015.
- [100] Abhilash Jindal, Abhinav Pathak, Y. Charlie Hu, and Samuel Midkiff. Hypnos: Understanding and treating sleep conflicts in smartphones. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 253–266, New York, NY, USA, 2013. ACM.

ABSTRACT**USER-CENTRIC POWER MANAGEMENT FOR MOBILE OPERATING SYSTEMS**

by

HUI CHEN**December 2015****Advisor:** Dr. Weisong Shi**Major:** Computer Science**Degree:** Doctor of Philosophy

The power consumption of mobile devices must be carefully managed to provide a satisfied battery life to users. This target, however, recently has become more and more difficult to complete. We still cannot expect the battery life problem be solved economically shortly, even though researchers already addressed many aspects of this problem. Principally, that's because existing power management systems, which concentrate on controlling hardware power states, cannot effectively make these hardware components work in low-power mode. Why is this the case?

Based on our analysis of 14 users' device usage trace, we found that background applications generate too many activities when the device is either idle or active. These activities are either unimportant or unnecessary for the user. However, a significant amount of CPU time was consumed by them. Moreover, these application activities cause many system services to consume a considerable quantity of battery energy. When we install more applications on our mobile devices, this situation will become even worse. Most application developers rarely consider the power consumption of applications. How to control application state and eliminate redundant application activities become more and more important. Existing power management systems, apparently, cannot handle this situation.

Some publications already tried to solve the problem several years ago. For example, EcoSystem and Cinder operating systems try to allocate battery energy precisely to applica-

tions based on their requirements. However, the problem with their solution is that the estimated application power consumption cannot accurately represent its reasonable demand. Energy-aware adaptation is another solution to decrease application power consumption. In our previous research, we implemented the *Anole* framework to supply energy adaptation APIs to applications. To use this framework, application developers have to implement power-saving strategies in their program. In the operating system, we need to change application behavior automatically in energy adaptation mode. We noticed the latest iOS operating system implemented the idea; the system notifies users to turn off background application update when the battery level is lower than 20%. However, this kind of uniformity in power management can hardly be accepted by most users, because user habits are different from each other.

We need to customize the power management strategy for each user. Otherwise, the user experience may be significantly impacted. To solve this problem, we propose user-centric power management, which utilizes the usage pattern of the individual user to distinguish important application from regular applications. Energy-saving strategies will not influence important applications to the user. From the analysis of 14 users' device usage traces, we found that most users' user behavior follows their pattern, which is both time-dependent and location-dependent. Based on this observation, we propose the *UPS* power management, which collects user behaviors and analyzes the usage pattern of users. We can easily use it to bridge usage behavior to energy-saving strategies. We also proposed three energy-saving strategies, *UCASS*, *LocalLite* and *WakeFilter*, to optimize the redundancy in background application activities and location service usage, and the abuse of in wakelock usage. Our simulation result based on real device usage traces shows that these three strategies can effectively save battery energy consumed background application activities, location requests, and wakelock requests.

AUTOBIOGRAPHICAL STATEMENT**HUI CHEN**

Hui Chen joined the Ph.D. program at Wayne State University in Jan 2010. He received his Master degree in Computer Science in Beijing Institute of Technology in Aug 2008 and received his Bachelor degree in Automation from Xidian University in Aug 2004. His research interests include Power Profiling, Energy-aware System Design, Sustainable Computing, and he has published several papers in workshops, conferences and journal, such as SUSCOM, IGCC, PASA, PMP. He has also served as a peer reviewer for many conferences and journals.