저작자표시-비영리-변경금지 2.0 대한민국

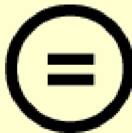
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사학위논문

스마트폰을 위한 사용자 중심 최적화 기법

User-Centric Optimization Techniques for Smartphones

2016년 8월

서울대학교 대학원
전기·컴퓨터 공학부
송욱

Abstract

Recently, smartphones have become an integral part of everyday life. In addition, as smartphone users are expecting their devices to deliver PC-level user experience, numerous design requirements are rapidly emerging with the technology development. In order to meet the demanding system design requirements, many conventional techniques, whose basic concepts are almost same as those of the traditional computing devices such as PCs, are applied to the smartphones. However, as truly personalized and interaction-oriented devices, the smartphones have distinct characteristics which distinguish the devices from traditional computing devices. Therefore, it is highly required to understand and to analyze the distinctive inherent characteristics of smartphones for a provision of a new novel opportunity for system optimization.

In this dissertation, we propose new user-centric optimization techniques to satisfy various design requirements of smartphones such as energy efficiency, effective thermal management and rapid responsiveness without any performance degradation by taking advantage of high-level information from the smartphone users.

We first introduce a new definition of the response time, the user-perceived response time, which is known to be a critical metric for the quality of user experience of the smartphone. We also present a user-perceived response time analyzer for Android-based smartphones, which can identify the user-perceived response time of smartphone apps during run time. Based

on on-line identification of the user-perceived response time, we propose a novel CPU power management framework, which enables more aggressive low-power techniques to be employed while executing display-insensitive parts of task executions. Our experimental results on a smartphone development board show that the proposed technique can reduce the CPU energy consumption by up to 65.6% over the Android's default `ondemand` *cpufreq* governor.

Second, we propose a novel dynamic thermal management (DTM) technique for smartphones, which ensures the quality of user experience during the execution of display-sensitive parts without any thermal violations. In the proposed DTM technique, in order to identify that the current execution could affect the visible portion of the display, we develop a user-perceived response time prediction model for each interactive session based on statistical analysis of the user-perceived response times for the past interactive sessions. By exploiting the on-line prediction of the user-perceived response time, the proposed DTM technique carefully makes the DTM decisions for a higher quality of user experience. Our experimental results on an ODROID-XU+E board show that the proposed technique can improve the user-perceived performance by up to 37.96% over the Android's default DTM policy.

Third, we present a personalized optimization framework for smartphones which can provide valuable high-level hints for optimizing the smartphone design requirements. The main goal of the proposed framework is collecting an app usage log of a smartphone user and analyzing the collected log so that particular usage patterns, if any, can be effectively identified. In

order to identify app usage patterns, a couple of app usage models are also proposed. Based on the app usage models developed, we also propose a launching experience optimization which avoids unnecessary app restarts considering the detrimental effects of the restart on user experience from the perspective of performance, energy, and loss of previous state. Our experimental results on the Nexus S Android reference phones show that our proposed optimization technique can avoid unnecessary application restarts by up to 78.4% over the default LRU-based policy of the Android platform.

Based on the evaluation for each technique, we verified that the user-centric optimization techniques improve the quality of user experience in terms of energy efficiency, effective thermal management and rapid responsiveness over previous system-centric techniques.

Keywords: Mobile System, Smartphone, Mobile Operating System, System Software, User-Centric Optimization, User Experience, Power, Thermal

Student Number: 2009-30920

Contents

I. Introduction **1**

1.1 Motivation 1

1.1.1 Distinctive Characteristics of Smartphone 1

1.1.2 Existing Optimization Techniques for Smartphones and Their Limitations 4

1.2 Dissertation Goals 7

1.3 Contributions 8

1.4 Dissertation Structure 9

II. Related Work **11**

2.1 Power Management Techniques 11

2.2 User Behavior Characterization 13

2.3 Launching Time Optimization Techniques 14

III. CPU Power Management Technique Using User-Perceived Response Time Analysis **19**

3.1 Motivation 19

3.2 Design and Implementation of URA 24

3.2.1 Overview 24

3.2.2 User-Perceived Resoponse Time Identification . . . 26

3.2.3 URA-based CPU Power Optimization Technique . . 29

3.3 Experimental Results 30

IV. SmartDTM: Smart Thermal Management for Smartphones 37

4.1 Overview 37

4.2 Motivation 40

4.3 Design and Implementation of SmartDTM 44

4.3.1 Basic Idea 44

4.3.2 Architectural Overview 47

4.3.3 User-Perceived Response Time Prediction 50

4.3.4 Worst-Case Temperature Estimation Model 52

4.4 Experimental Results 54

4.4.1 Experimental Environment 54

4.4.2 Performance Evaluation 56

4.4.3 Temperature Evaluation 58

V. Personalized Optimization Framework 65

5.1 Motivation 65

5.2 Design and Implementation of POA 66

5.2.1 Design Overview 66

5.2.2 App Usage Modeling Module 70

5.2.3 Usage Model-Based Optimization Module 71

5.3 App Usage Model Construction 71

5.3.1 P-AUM: Pattern-based App Usage Model 71

5.3.2 C-AUM: Clustering-based App Usage Model 75

VI. AUM-based Launching Experience Optimization Technique 84

6.1 Motivation 84

6.1.1 Impact of Cold Starts on App Launching Experience 86

6.1.2 Android Task Management Scheme 89
6.1.3 Problem of the LRU-based Task Killer 90
6.2 AUM-based Launching Experience Optimization 94
6.2.1 App Usage (AU)-aware Task Killer 94
6.2.2 App Usage (AU)-aware Prelauncher 95
6.3 Experimental Results 97
6.3.1 Experiment Environment 97
6.3.2 Results of Task Killing Mechanism Optimization . . 98
6.3.3 Results of Prelaunching Technique 106

VII. Conclusions 107
7.1 Summary and Conclusions 107
7.2 Future Work 109
7.2.1 Improving Prediction Accuracy of the AUMs Using Context Information 109
7.2.2 Integrated Intra- and Inter-App Approaches for User-Centric Optimizations 111
7.2.3 User-Centric Optimizations for The Other Design Requirements 112

Bibliography 114

List of Figures

Figure 1. A life cycle diagram of the smartphone app. 6

Figure 2. Distributions of survey participants according to their user experience for six apps [20]. 12

Figure 3. An architectural overview of the FALCON system [16]. 17

Figure 4. Two cases on how S_i is divided into $I_{S_i}^{perc}$ and $I_{S_i}^{oblv}$. . . 20

Figure 5. Changes in the CPU power consumption during the app launching session. 21

Figure 6. CPU energy consumption breakdowns between I^{perc} and I^{oblv} over varying durations of the think time. . . . 23

Figure 7. An architectural overview of **ura**. 26

Figure 8. An example of identifying the user-perceived response time. 29

Figure 9. User-perceived response time differences between **ura** and manual measurements. 33

Figure 10. Changes in the average energy saving of 14 interactive sessions over varying durations of the think time. . . . 34

Figure 11. An illustrative example of the difference between the Android's default DTM policy and the no-DTM policy with the `oninterval` *cpufreq* governor. 42

Figure 12. Normalized user-perceived response time comparisons between the no-DTM and default DTM policies. 43

Figure 13. An overview of the default DTM policy with the `interactive` *cpufreq* governor. 46
Figure 14. Examples of how the SmartDTM restricts the DTM decisions or not using on-line estimation of t_{end} and $t_{critical}$. 48
Figure 15. An architectural overview of SmartDTM. 49
Figure 16. A GUI example of the **twitter** app and its VIEW hierarchy. 51
Figure 17. Changes in the CPU power consumption and temperature under heavy usage scenarios. 53
Figure 18. Differences of the average CPU power consumption and the elapsed times corresponding to the CPU temperature changes from 75 °C to 85 °C between the various workload characteristics. 55
Figure 19. A comparison of normalized user-perceived response times for 10 launching interactive sessions when the base temperature is 65 °C. 58
Figure 20. A comparison of normalized user-perceived response times for 17 interactive sessions when the base temperature is 70 °C. 59
Figure 21. The average and maximum CPU temperature differences between the SmartDTM and default policies when the base temperature is 65 °C. 61

Figure 22. The average and maximum CPU temperature differences between the SmartDTM and default policies when the base temperature is 70 °C. 62
Figure 23. Distributions of the elapse times from the time when the execution of the user-perceived response time interval ends to the time when the current temperature drops to 65 °C. 64
Figure 24. Per-user app usage distribution. 67
Figure 25. Per-app frequency as one of three next launched apps right after each *Memo App* launch (Total 38 *Memo App* launches). 67
Figure 26. An architectural overview of the proposed POA framework. 68
Figure 27. An example of building an app usage model. 68
Figure 28. An example of using an app usage pattern in optimizing the LRU-based task killing policy. 69
Figure 29. An example of how the P-AUM heuristic finds apps to be launched next. 73
Figure 30. An example of the launch radius and the clustering affinity between two apps. 77
Figure 31. An example of building clusters using the *single-linkage clustering* algorithm. 79
Figure 32. Launching time differences between hot and cold starts. 86
Figure 33. Current changes during the launching process. 89

Figure 34. A breakdown of 60 apps based on the degree of state preservation. 89
Figure 35. Restart ratio distributions of 21 users. 91
Figure 36. Changes in LRU stack positions of *App A* over time. . . 93
Figure 37. The overview of the AU-aware prelauncher. 96
Figure 38. Restart ratio comparisons of five policies. 99
Figure 39. Changes in the rank of *App A* over time. 100
Figure 40. Weighted restart count comparisons of five policies. . . 101
Figure 41. Normalized launching time comparisons of four policies. 103
Figure 42. Normalized energy consumption comparisons of four policies. 103
Figure 43. State loss ratio comparisons of four policies. 104
Figure 44. An example of computing termination accuracy grades. 105
Figure 45. Comparisons of the average termination accuracy grade of four policies. 105
Figure 46. The effect of the prelaunching technique on the restart ratio. 106

List of Tables

Table 1. Scenario descriptions of 7 benchmark apps. 32

Table 2. A comparison of normalized user-perceived response times of seven second interactive sessions over varying durations of the think time in the first interactive sessions. . . 35

Table 3. Summary of the workloads in the *micro_bench* binary. . . 55

Table 4. Scenario descriptions of 10 apps used in the experiments. 56

Table 5. Summary of four representative user logs. 98

Chapter 1

Introduction

1.1 Motivation

Recently, smartphones have become an integral part of everyday life. Since most smartphone users expect their smartphones to be always-on, always-connected devices and their attention is also severely limited [1, 2], the immediate responsiveness of the smartphone to user inputs is an important design requirement that affects the quality of user experience. At the same time, for the smartphones, since the batteries of smartphones are limited in size and therefore capacity in general, energy efficiency is also a major design requirement. Furthermore, in order to achieve the demanding requirements of the rapid responsiveness and the low-power operation, modern mobile processors have adopted multi-core architectures while increasing the operating frequency, resulting in an increase in the power density of the mobile processor. Therefore, the effective thermal management is rapidly emerging as one of the important design requirements.

1.1.1 Distinctive Characteristics of Smartphone

Smartphones have distinct characteristics which distinguish them from traditional computing devices such as PCs. First, smartphones are highly interaction-oriented devices because most of the usage scenarios on smart-

phones involve frequent user interactions with smartphones. Moreover, in most cases, a user tends to focus on one app at a time although multi-tasking support is commonly available for modern smartphones [5]. For example, when reading new incoming emails, the user first launches an email app. Once the email app is launched, the user opens an email and reads it for a while. In this example, the user's whole attention is directed to a single interactive session at a time. Therefore, the quality of user experience with smartphones largely depends on how smoothly and quickly smartphones react to the user's various interactions. In the case of an email app, the quality of user experience is significantly affected by the launching time (which can be regarded as the response time of an app launching interaction) and the response time of loading a selected email. Since the response time of an interactive session has a large impact on the quality of user experience, understanding and analyzing the response time of an interactive session in smartphones are important requirements for improving user experience.

In conventional computing systems, the response time of a task is defined as the length of the time interval between the start and the end of a task execution. However, for smartphones, most users have a tendency to interact with the smartphones in a hurried fashion so that they *subjectively* decide that the smartphones are ready for the next interaction even though the task execution has not been fully completed. For this reason, the existing definition of the response time, which we call *computation-centric*, is not appropriate for accurately representing the user-perceived effective response time in smartphones [6]. For example, in the case of launching an email app, a user may consider that the launching has been completed

when the visible user interface for the next interaction appeared on the display as its final form even though several display-insensitive computations may be still executing. In this case, the *user-perceived* response time is a lot shorter than the computation-centric response time. Therefore, in order to accurately represent the user-perceived response time in smartphones, we need a different definition of the response time from the smartphone user's perspective.

Second, the smartphone is also a truly personalized computing device closely connected with only one dominant user. Motivated by this fact, smartphones present a unique new opportunity for system optimization. A possibility of this type of *personalized optimization* is supported by various smartphone usage studies. For example, a recent study [3] shows that only a small number of different apps[1] are used by each user, although there are more than 200,000 apps available from the Android app market [4]. In our own study on the Android app usage profiling of 21 college students and engineers, we have observed a similar tendency. In our study, each user has used, on average, 52 apps over the period of two weeks. Furthermore, we have also observed that, for most users, there is a small set of distinctive app usage patterns that are repeatedly appearing. In particular, it was quite common to see that two apps are strongly related each other, often being launched successively. For example, one of our 21 study participants has launched *Memo App* 38 times and *Media Player App* 96 times of total 2454 app launches in a period of two weeks. Furthermore, in more than 70% of *Memo App* launches, *Media Player App* was launched as one of next three

[1] In this dissertation, we call an application by a (more popular) shorthand, app.

apps launched following *Memo App*. If we could identify these distinctive usage patterns among a small set of favored apps during run time, we can improve, for example, user experience by giving these apps favors.

1.1.2 Existing Optimization Techniques for Smartphones and Their Limitations

The basic strategy for higher energy efficiency in the smartphone is to put the system into a sleep or a suspend mode as soon as possible. In addition, for the dynamic power optimization, the dynamic voltage and frequency scaling (DVFS) technique is widely applied to various components of the smartphone, such as the CPU, GPU, and memory interface. The main purpose of the DVFS technique is to select the operating frequency as low as possible while still providing acceptable performance for a higher efficiency. In the conventional DVFS techniques, the utilization of the target devices is commonly used to account for the performance demands of tasks. However, as mentioned in , an execution of a given user-interactive session can be divided into two intervals, one where the system response time directly affects user experience and the other where the system response time does not affect user experience. Therefore, for the smartphone, it is required to consider whether the current execution is in the interval where the system response time directly affects user experience or not as well as the utilization of the target device.

For the smartphones, due to the limited space of their form factors, it is difficult to apply the mechanical solutions, such as a fan-cooled heat sink

and a liquid cooling device, to control the temperature of the target device. Instead, software-based techniques are the most common solutions to handle the emerging thermal problems in the smartphones. The dynamic thermal management (DTM) scheme is one of the widely adopted software-based thermal management techniques to the smartphone. The basic idea behind the DTM scheme for the CPU is to aggressively applying CPU throttling when the over-heating is detected. Since the power consumption of the device is highly related to the temperature, the conventional DTM scheme can be an effective solution for the thermal problems. However, as a side effect, the performance degradation may occur while applying the DTM scheme. Therefore, since there is a specific execution interval, whose quality of user experience is highly affected by the system performance level, the DTM scheme should also consider the impact of applying CPU throttling during the execution of such interval.

In order to quickly respond to user inputs, on most smartphone systems, such as iOS, Android, and Windows Phone, only one app is allowed to run in the foreground at any given moment on most smartphone systems. Therefore, it is possible to ensure the app which the user is mainly focusing on has the sufficient resources to provide a smooth and responsive experience. Furthermore, apps in the smartphone also have a unique life cycle which is completely different from that of the desktop and the server environment. Figure 1 illustrates how the execution state changes during a smartphone app's life cycle. When the user selects an app in the not running state from the app launcher, a new instance of the app is created. After being launched, with the user-visible interface for the next interaction appeared

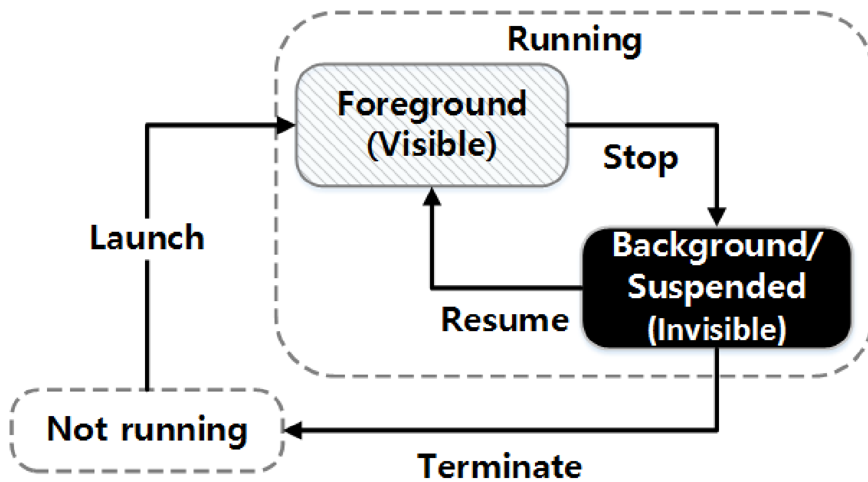


Figure 1: A life cycle diagram of the smartphone app.

on the display, the app is in the foreground state. When the user presses the Home button in order to stop the foreground app, the state is changed from the foreground state to the background or suspended state. In this case, for better app launching experience, the system does not immediately terminate the app. Instead, such apps are kept as the background or suspended apps in the main memory of smartphones, thus, they can be quickly responded when the apps are launched again (that is, resumed) in the future. Those background or suspended apps are evicted from the memory (that is, terminated) when the systems decide that they need more memory for, say, a new app. However, in most cases, the least recently usage (LRU)-based policy is applied to select those victim apps. Considering that there are remarkable personalized features in the usage of the smartphone, it is obvious that such technique cannot adapt to changing user contexts since the LRU-based policy only considers the recency of the app usage.

1.2 Dissertation Goals

In this dissertations, we propose new user-centric optimization techniques to satisfy various design requirements of smartphones such as energy efficiency, effective thermal management and rapid responsiveness without any performance degradation by taking advantage of high-level information from the smartphone users. The primary goals of the dissertation are as follows:

- Presenting a user-perceived response time analyzer for Android-based smartphones which can identify the display-centric response time of smartphone apps during run time.
- Proposing novel CPU power and dynamic thermal management techniques based on the on-line identification of the display-centric response time.
- Presenting a personalized optimization framework for smartphones which can provide valuable high-level hints for optimization of various smartphone design requirements.
- Proposing an app launching experience optimization technique which can avoid unnecessary app restarts based on the proposed personalized optimization framework.
- Evaluating the effectiveness of the proposed techniques by comparing to the conventional techniques.

1.3 Contributions

The proposed user-centric approach in this dissertation adds a new dimension to improve user experience of the smartphones as follows:

- We introduced a new definition of the response time, which we call *the display-centric response time*, which is known to be a critical metric for the quality of user experience of the smartphone. In order to identify the display-centric response time of smartphone apps during run time, we developed **ura**, a **u**ser-perceived **r**esponse time **a**nalyzer for Android-based smartphones.

- We proposed a novel CPU power management framework based on on-line user-perceived response time analysis. Based on **ura**'s on-line identification of the display-centric response time, our proposed framework enables more aggressive low-power techniques to be employed while executing display-insensitive parts of task executions (which do not affect the user-perceived response time). As a concrete example of low-power techniques, we use dynamic voltage scaling (DVS) to demonstrate our proposed technique.

- We proposed a new dynamic thermal management technique for smartphones, called SmartDTM. By also taking advantage of **ura**'s on-line identification of the display-centric response time, our proposed SmartDTM ensures the quality of user experience during the execution of display-sensitive parts without any thermal violations which are predefined by the system.

- We present the design and implementation of a personalized optimization framework for Android smartphones, called POA. The main function of the POA framework is to collect an app usage log of a smartphone user and to analyze the collected log so that particular usage patterns, if any, can be effectively identified. In order to identify app usage patterns, we developed a couple of app usage models (AUMs).
- Based on the AUMs developed, we proposed a launching experience optimization which avoids unnecessary app restarts considering the detrimental effects of the restart on user experience from the perspective of performance, energy, and loss of previous state.
- We implemented the proposed framework and techniques in the Android platform and Linux kernel. Then, we evaluated their effectiveness compared to the conventional techniques.

1.4 Dissertation Structure

This dissertation consists of seven chapters. The first chapter presents a introduction to this dissertation while the last chapter serves as a conclusion with a summary and future work. The five intermediate chapters are organized as follows:

Chapter 2 introduces the existing optimization techniques closely related to this dissertation.

Chapter 3 explains the key idea behind our proposed CPU power optimization framework and describes an overview of **ura** and illustrate how the

ura-based CPU power management technique can improve the CPU energy efficiency.

Chapter 4 presents a dynamic thermal management technique for smartphones (SmartDTM). We, first, show the limitation of the conventional dynamic thermal management technique, which is widely applied to the smartphones. And then, we explain how the proposed dynamic thermal management technique can improve the quality of user experience without any thermal violations.

Chapter 5 describes an overview of the proposed POA framework and illustrates how the POA framework can be utilized to improve the launching experience using a small example. Two proposed app usage models are also explained in this chapter.

Chapter 6 presents an app launching experience optimization technique based on the AUMs developed. In order to better motivate our proposed optimization technique, we also present quantitative analysis of the impact of cold starts on performance, energy, and state preservation in this chapter. effect of the proposed technique on NAND endurance is presented in detail.

Chapter 2

Related Work

2.1 Power Management Techniques

Several groups have proposed CPU frequency-scaling techniques by taking account of the quality of user experience. In particular, SmartCap [20] has shown that the neural network-based inference model can be useful to decide the minimal acceptable frequency without degrading user experience. In order to avoid frequency over-provisioning, SmartCap takes a frequency-capping approach in adjusting the CPU frequency. In detail, the objective of SmartCap is to find the saturated frequency, which still guarantees the quality of the user experience close to that at the highest frequency. In order to better motivate their proposed power management scheme, they conducted a statistical sample survey. They investigated the impact of the CPU frequency on the satisfaction of the quality of the user experience. A distribution of the number of participants regarding their user experience, which is subjectively measured by the participants varying the CPU frequency, is shown in Figure 2. Intuitively, it is obvious that the proportion of the participants grading *Good* should increase with the higher frequency, while that of *Poor* should decrease, as Talking Tom, Snow Pro, and Storm show. However, in the case of Fruit Ninja and UC Browser, it was shown that the higher frequency than 600 MHz does not improve the quality of the user

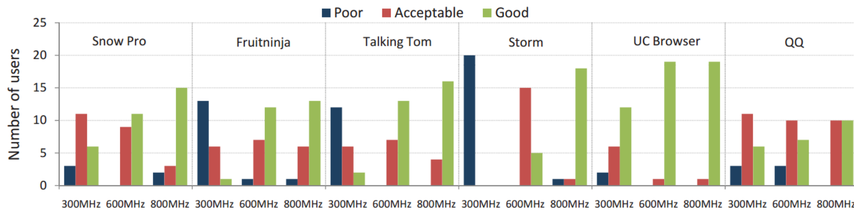


Figure 2: Distributions of survey participants according to their user experience for six apps [20].

experience. Therefore, the *saturated frequency* can be defined as the maximum frequency beyond which the quality of the user experience is slightly improved. Based on this observation, they built an inference model, which predicts the saturated frequency of an app. The inference model relies on the system activity features correlated with the saturated frequency. SmartCap is designed with a server-client architecture. The server is responsible for the model training, then provides the inference parameter to the client. At the client-side, based on the prediction result from the inference model, SmartCap can adaptively restrict the possible highest frequency of an app.

The technique proposed in AURA [21] is also similar to our approach. In their approach, AURA can effectively decide the CPU frequency for each interactive session by exploiting an app classification scheme based on the user interaction intensity. They propose CPU frequency setting algorithms based on the Markov Decision Process using the app classification scheme. However, our work is fundamentally different from existing CPU frequency scaling techniques in that we take advantage of the user-perceived response time as a main hint for adjusting the CPU frequency.

Our proposed technique can be viewed as a simplified version of vertically-

integrated OS-directed power management techniques such as Application Modes [22]. In Application Modes, for example, an app provides several working modes with different data fidelity while consuming different amount of power. Using a narrow interface between the OS and apps, during run time, the OS informs the app of mode transitions by which the limited battery capacity can be better managed. The OS makes such a transition taking account of vertically-collected information such as user preference, available application modes and the remaining battery capacity. By interpreting as a mode transition event the detection of the end of the user-perceived response time interval, which signifies an important fidelity change point from the user's perspective, our approach can be viewed as a simplified version of Application Modes. However, our proposed approach does not require any effort from app developers, which we believe the main advantage of our approach over Application Modes.

2.2 User Behavior Characterization

From the earlier days of smartphones, many researchers have recognized that understanding smartphone user's behavior and their interaction patterns with smartphones will be important in creating *smarter* applications. Therefore, several groups have conducted usage studies for characterizing how smartphones are used [7, 8, 9, 10]. For example, Shye *et al.* [7] observed that only a few states and transitions are required to build a user activity model on the smartphone usage behavior. Falaki *et al.* [10] characterized smartphone usage in terms of user activities and their impact

on network and battery, by analyzing detailed usage traces from 255 users. From their analysis, Falaki *et al*. suggest that, because of diverse usage profile differences, mechanisms designed to average case behaviors are likely to be ineffective. Rather, they demonstrated that user-specific learning and adaptation is a more effective approach.

The observation from our smartphone app usage study also agrees with the findings of Shye *et al*. [7] and Falaki *et al*. [10]. However, to the best of our knowledge, our work is the first attempt to integrate personalized optimization into real systems. Furthermore, our work is also quite different from previous efforts in that we focus on system-level optimizations.

2.3 Launching Time Optimization Techniques

There have been many efforts to reduce the application launching time in both non-mobile and mobile computing systems. In particular, in order to hide the access-time gap between the main memory and the hard disk drive (HDD), prefetching techniques have been extensively studied. However, since NAND flash memory has been widely used as a main storage device for most smartphones, existing launching time optimizations [11, 12] for HDDs cannot be directly applied to smartphones.

Recent investigations have more directly focused on improving application launch performance on NAND flash-based storage devices, such as solid state drives (SSDs) [13, 14]. For example, Joo et al. presented an SSD-aware application prefetching scheme, called FAST [15]. FAST exploits the fact that the I/O time can be overlapped with the computation time dur-

ing the application launch procedure. While these traditional techniques are effective in reducing the launching time by intelligently exploiting the underlying devices' characteristics, more advanced solutions that take advantage of high-level information (such as app usage patterns) in optimizing the launching time is highly desirable.

Yan et al. proposed an app launching time optimization technique, called FALCON [16]. In FALCON, based on the prediction about the single user's future app launches, the apps, which are more likely to appear in the future, are prelaunched to improve the app launching experience. In order to identify features which give strong insight into the next app to be launched, they analyzed trace data collected from the Rice LiveLab user study [17] and their own internal user study. From this data analysis, they derived three following personalized features. First, certain apps are more likely to be the *trigger apps* which start sessions than others, while other apps are more likely to be *follower apps* which are launched in the same session after using the trigger apps. Second, there is a particular location strongly correlated with distinctive app usage patterns. For example, one of their study participants has shown a tendency that the Game apps are heavily used at home while the Browser and Calendar apps are mostly used at work. Third, a user's app usage pattern can be dramatically changed over longer time period. In addition, their analysis reveals that the durations of such intensive usage are varied depending on the type of each app. In particular, the Game apps tend to have more intense usage during shorter time period in comparison with those of other apps. Figure 3 shows an overview of the FALCON system. The launch predictor decides apps which are likely to be

launched in the future using the context information. It consists of four submodlues: the feature extractor, the decision engine, the model trainer, and the proc tracker. The feature extractor is responsible for converting raw context data from the context source manager into the personalized features for the decision engine and the model trainer. Based on the personalized features, the multi-feature decision engine of the FALCON system determines which features to use and what apps to prelaunch. When the prediction is made by the decision engine, it is passed to the dispatcher, which initiates the selected apps and performs the prelaunch routine of the apps.

As shown in their experimental results, even if the training data size is only 42 days long, over 70% of the targeted apps can be proactively loaded before their actual launches. This is because the personalized features used in the proposed inference model are strongly indicative of the app usage patterns. However, considering that the prediction should be made in the smartphone environment, conducting inference using all those personalized features, which give direct insight into the future app usage, is computationally expensive. In order to handle this problem, the current implementation of FALCON relies on external servers for model training. On the other hand, we take a more simplified approach to build the app usage models. Our proposed approach does not require any of computation-intensive or privacy-sensitive context features such as the location information but exploits only a past app usage log in prediction. In addition, by exploiting relatively simpler methods than the multi-feature inference model, it is possible to build the app usage models dynamically in the smartphone environment. We also showed that our proposed app usage models can provide proper prediction

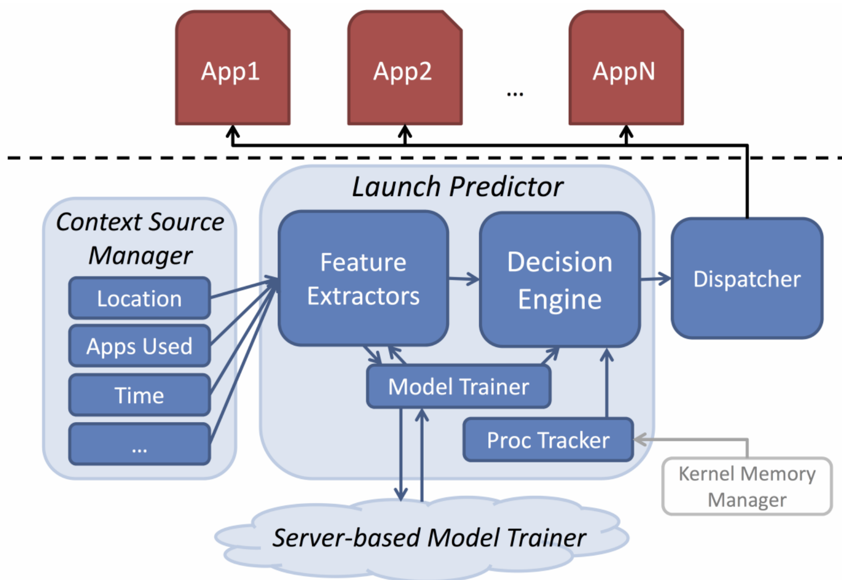


Figure 3: An architectural overview of the FALCON system [16].

accuracy for the launching time optimization without any support of the external servers.

Although several algorithms have been proposed to predict future app launches using various types of user context information, they only focused on solving the problem that which app will be used next rather than when it will be used. Considering that it is significantly important to instantly provide fresh content for a better user experience, it is also highly required to predict not only which app is more likely to be used but also when it will be used. Therefore, Parate et al. [18] developed two models, a prediction model, called App Prediction by Partial Match (APPM), which decides an app which is likely to be used next, and a Time Till Usage (TTU) temporal model, which determines the moment when the predicted app should be

used. Based on the models developed, they also proposed a preeminently practical approach to prefetch, called PREPP, which prefetches the app content at the appropriate moment to improve the user experience.

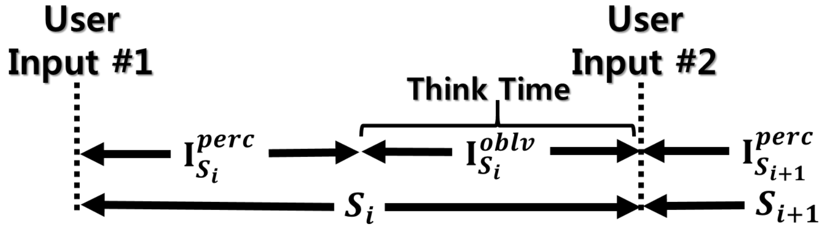
Although previous efforts such as FALCON and PREPP showed that it is possible to improve the app launching experience by employing only the prelaunch-based techniques, the cached apps, which already reside in the main memory, are do not considered at all. In order to fully achieve a higher app launching experience, it should be also considered how the cached apps can be effectively managed. In contrast to previous works, our proposed approach is different in that we focus on both the termination and prelaunch mechanisms simultaneously to optimize the app launching experience.

Chapter 3

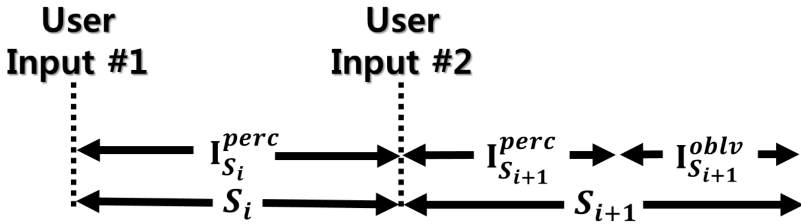
CPU Power Management Technique Using User-Perceived Response Time Analysis

3.1 Motivation

As a truly interaction-oriented device, most usage scenarios of a smartphone are composed of a sequence of *interactive sessions*, S_1, ..., S_N. Each interactive session S_i is defined as an interval between two consecutive user inputs. We can further divide the execution of an interactive session S_i into two subintervals, $I_{S_i}^{perc}$ and $I_{S_i}^{oblv}$, a *user-perceived response time interval* and a *user-oblivious response time interval*, respectively. $I_{S_i}^{perc}$ represents the period from the beginning of the interactive session S_i initiated by a certain user input to the time when all of the user-visible interface for the next user interaction are displayed. (In other words, the length of $I_{S_i}^{perc}$ is the display-centric response time of S_i.) $I_{S_i}^{oblv}$, on the other hand, corresponds to the user's think time before the next user interaction. (The length of $I_{S_i}^{oblv}$ is determined by the time when the next user input is entered for the next interaction.) Figure 4 illustrates how the interactive session S_i may be further divided into $I_{S_i}^{perc}$ and $I_{S_i}^{oblv}$. In the case of the example in Figure 4(a), after all the user-visible contents are drawn for the user input #1, S_{i+1} is



(a) The case when S_i is divided into $I_{S_i}^{perc}$ and $I_{S_i}^{oblv}$.



(b) The case when S_i consists of $I_{S_i}^{perc}$ only without any think time.

Figure 4: Two cases on how S_i is divided into $I_{S_i}^{perc}$ and $I_{S_i}^{oblv}$.

initiated after some think time (i.e., after $I_{S_i}^{oblv}$) by the user input #2. On the other hand, the user may input the user input #2 right after $I_{S_i}^{perc}$ without any think time.[1] Figure 4(b) shows such an example. The length of $I_{S_i}^{oblv}$ is almost zero in this case and the new interactive session S_{i+1} begins without any think time. Since the system performance level in I_S^{oblv} is less likely to affect the quality of user experience, we may take a more aggressive approach in optimizing power/energy consumption while executing in I_S^{oblv} without degrading the quality of user experience.

In order to better motivate our proposed optimization framework, we illustrate how the energy consumption of an app launching interactive ses-

[1]There is the third case when the user input #2 is initiated within $I_{S_i}^{perc}$. Since our proposed technique does not change the execution behavior in $I_{S_i}^{perc}$, it is considered as the beginning of the new interactive session S_{i+1}.

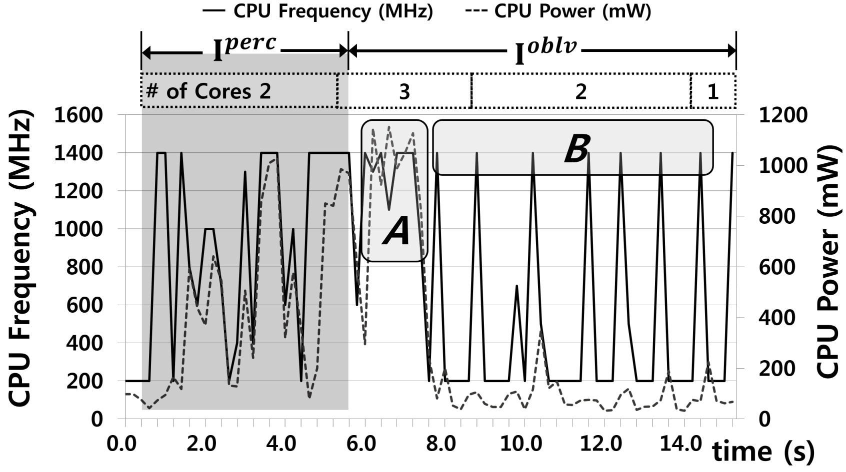


Figure 5: Changes in the CPU power consumption during the app launching session.

sion S_L can be improved using an example. Figure 5 shows how the CPU power consumption changes during the first 15 seconds after the Android `web browser` is launched. In order to emphasize differences between $I_{S_L}^{perc}$ and $I_{S_L}^{oblv}$, we chose the mobile start page of Yahoo (which included a fair amount of background computations in $I_{S_L}^{oblv}$). The X-axis, the Y-axis on the left side, and the Y-axis on the right side represent the elapsed time, CPU frequency, and CPU power consumption, respectively. In this example, at t $= 0.4$, the `web browser` app is launched. After the required resources are downloaded from the Yahoo website, all the user-visible contents are drawn at t $= 5.6$. That is, $I_{S_L}^{perc}$=$[0.4, 5.6]$. As shown in Figure 5, although there are additional computations related to the launching session, the user perceives that the launching of `web browser` was completed at t $= 5.6$. (We will describe how to identify the end of $I_{S_L}^{perc}$ in Section 3.2.2.) Even when exe-

cuting in $I_{S_L}^{oblv}$, we observe that the CPU frequency is frequently increased to the maximum frequency of 1,400 MHz. Furthermore, the number of active cores remains three[2] until t $= 8.6$. In detail, `web browser` executes tasks such as storing the downloaded resources in the web cache, capturing a snapshot of the current web page, and updating the browser history (as shown in the area A.) In addition, in the area *B*, `web browser` regularly renders Javascript code in the web page even though the web page does not dynamically change the displayed contents [26]. Since the execution in $I_{S_L}^{oblv}$ does not affect the quality of user experience, if we knew that the current execution were in $I_{S_L}^{oblv}$, we could have lowered the CPU frequency to the minimum frequency of 200 MHz. In this case, the energy consumption in $I_{S_L}^{oblv}$=$[5.6,\ 8.6]$ could be reduced by 47.0% over the Android's default CPU DVS policy, assuming that the next user input event occurred after the think time of 3 seconds. This is because developers tend to give a high priority to the tasks of displaying user-visible interfaces to quickly react to the user's input. Therefore, such task execution in I_S^{oblv} is widely observed for interactive sessions of other apps. For example, in the case of open source `twitter` apps, tasks such as storing the downloaded resources, caching a stream of tweets shown in the timeline, and synchronizing account information of the user are executed without affecting the changes in the user-visible contents.

[2]The SMDK board has a quad-core ARM cortex-A9 as a main CPU.

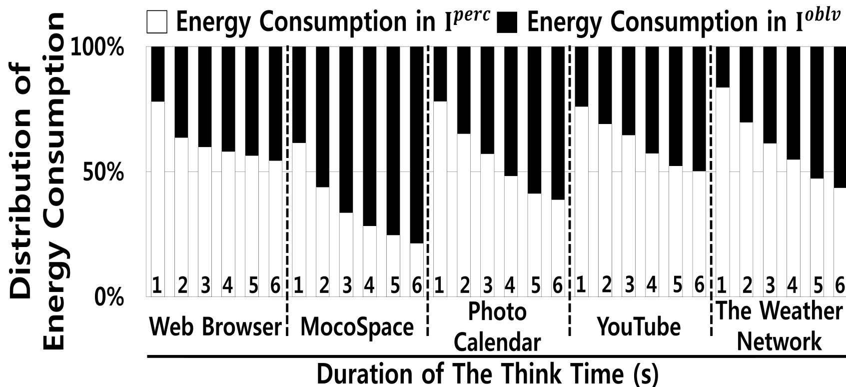


Figure 6: CPU energy consumption breakdowns between I^{perc} and I^{oblv} over varying durations of the think time.

In order to evaluate the applicability and effectiveness of our proposed framework over different apps, we also measured the energy consumed in both I^{perc} and I^{oblv}, varying duration of the think time in our SMDK measurement board. Figure 6 summarizes breakdowns of the CPU energy consumption between I^{perc} and I^{oblv} for five apps over different durations of the think time. We measured the CPU energy consumption during the launching sessions of the five apps. Our measurements show that up to 78.7% of the total CPU energy is consumed in I^{oblv}. For example, in the case of `web browser`, when the think time is set to 6 seconds, 45.7% of the total CPU energy is consumed in I^{oblv}. When the think time is set to 3 seconds, 46.6% of the total CPU energy consumption is from I^{oblv} on average. Since we can lower the CPU frequency more aggressively in I^{oblv}, our measurement result strongly indicates that our proposed framework can significantly reduce the energy consumption of smartphones.

3.2 Design and Implementation of URA

3.2.1 Overview

Our **ura**-based power optimization framework requires the user-perceived response time of each interactive session S_i to be computed on-line. In the proposed framework, **ura** is responsible for the identification of the end of $I_{S_i}^{perc}$ during run time from the execution of S_i. Our proposed aggressive optimization technique, therefore, can be immediately applied from the first $I_{S_i}^{oblv}$ execution. In this section, we describe the design and implementation of our **ura**-based power optimization framework for the Android platform.

In Android, only one UI thread per app is allowed to update all the user-visible contents of an app. Furthermore, when a display-update request is issued by the UI thread, the Android platform does not immediately re-draw the visible user interface. Instead, it is first posted to the event queue of the UI thread of the app and then, the UI thread subsequently dequeues the request and handles it. Exploiting Android's display update mechanism, **ura** can identify the end of $I_{S_i}^{perc}$ by tracking all the display-update requests related to the user interaction in a given interactive session S_i and detecting when the last display-update request of S_i is processed. Although the UI interaction with the user is the main source of generating display-update requests, it is not the only source. For example, in order to refresh user visible contents such as advertisement banners, display-update requests can be also generated. In this case, **ura** can automatically distinguish such requests from the display-update requests related to the user interaction because **ura** only tracks the display-update requests issued by the user input.

In order to detect the end of $I_{S_i}^{perc}$ for a given S_i, **ura** works as follows:

Step 1. Catch an interactive user input which indicates the beginning of S_i.

Step 2. Keep track of all the spawned threads from the user input (if any).

Step 3. Detect display-update requests related to serving the user input.

Step 4. Check whether all the display-update requests were processed so that the end of $I_{S_i}^{perc}$ can be decided. (In the case of when the next user interaction occurs before all the display-update requests are processed, the end of $I_{S_i}^{perc}$ can be also decided at this moment.)

Figure 7 shows an architectural overview of **ura** and the **ura**-based *cpufreq* governor within the Android platform. **ura** consists of two main modules, the modified method call interpreter, modInterpreter, and the end of user-perceived response time identifier, endIdentifier. As an additional module to the Dalvik VM, modInterpreter is responsible for steps 1, 2, and 3. For step 4, endIdentifier determines the user-perceived response time. By taking advantage of the user-perceived response time, the **ura**-based governor adjusts the CPU frequency to achieve a higher CPU energy efficiency. In the current implementation, the Davik VM interpreter is modified to instrument the method invocation and method return during run time. For this reason, **ura** cannot trace the native method invocations, which are included in native libraries such as the native OpenGL [3]. However, the OpenGL ES

[3] Although there are important apps (such as game apps) that use the native OpenGL, most Android apps are written in JAVA only.

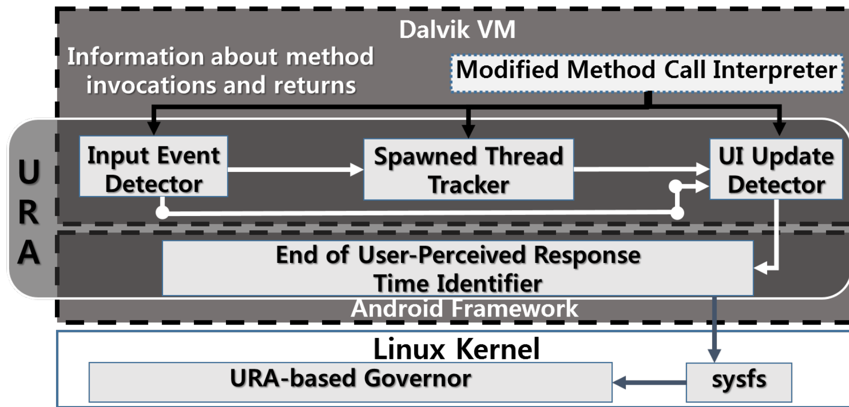


Figure 7: An architectural overview of **ura**.

API, which is based on the Java language and provided by the Android SDK, are fully supported by **ura**.

3.2.2 User-Perceived Resoponse Time Identification

ModInterpreter consists of three submodules, the input event detector, the spawned thread tracker, and the UI update detector. The main function of the input event detector is to capture events related to a particular user input. The spawned thread tracker is responsible for tracing newly spawned threads while processing the user input. All the message exchanges between the main thread and spawned threads are also traced by the spawned thread tracker. Besides, if the main thread sends messages to the other threads (which are already spawned before the user input), such threads are tracked as well. The UI update detector keeps track of display-update requests created for serving the user input.

Figure 8 illustrates how **ura** identifies the user-perceived response time

using an example. When a user interacts with Android UI components such as WIDGET and VIEW packages, a callback method in the event listener interface is invoked to handle a particular interaction. (In order to support different types of user interactions, the Android SDK provides various callback methods. For example, user interactions such as a touch, a click, and a long-click are handled by *onTouch()*, *onClick()*, and *onLongClick()* methods, respectively.) In the example in Figure 8, the callback method, *onClick()*, is called because the user clicks the user interface resource such as the BUTTON WIDGET. As the first step of identifying the user-perceived response time, the input event detector traces all the method invocations related to the callbacks for the user input, so as to identify the start t_s of the current interactive session S_i. In the case of the example in Figure 8, the input event detector catches the *onClick()* invocation. For steps 2 and 3, the input event detector also provides both the spawned thread tracker and the UI update detector with information about all the method invocations during the execution of *onClick()*.

When an app is launched, a special thread, called main thread, is created by the Android system. While only main thread can update the user-visible contents, compute-intensive work is performed by separate threads, called worker threads, for better responsiveness. If a worker thread requires updating the user interface, such requests are delegated to main thread. In order to support inter-process communication (IPC) between main thread and worker thread, various APIs are supported by the Android SDK, for interchanging *Message* and *Runnable* objects between the main thread and the worker thread. By exploiting the information (which is provided by

the input event detector) on the method invocations during the execution of the callbacks for the user input, the spawned thread tracker traces newly spawned worker threads and all the invocations among such IPC APIs. For example, as shown in step 2 of Figure 8, when main thread wants to perform compute-intensive work via worker thread, main thread invokes *sendMessage()* while the worker thread invokes *dispatchMessage()*. In this case, the spawned thread tracker catches the *sendMessage()* and *dispatchMessage()* invocations. And then, in order to detect UI update requests created by worker thread, information about all the method invocations during the execution of *dispatchMessage()* is fed to the UI update detector.

To recognize the changes in the user-visible contents, **ura** traces UI update requests issued by the user input and captures the moment at which the last request is handled. For example, at step 3 in Figure 8, the *invalidate()* methods are invoked twice during the execution of both *onClick()* and *dispatchMessage()*. At these points, the UI update requests are posted to the event queue of main thread. In order to track the UI update requests, the UI update detector thus catches the *invalidate()* invocations and watches the event queue for the UI update requests. Subsequently, when main thread dequeues the last update request from the event queue and invokes *draw()* to handle it (at t_e in step 4 of Figure 8), endIdentifier determines t_e as the end of I^{perc}. In this example, the user-perceived response time is estimated as (t_e-t_s).

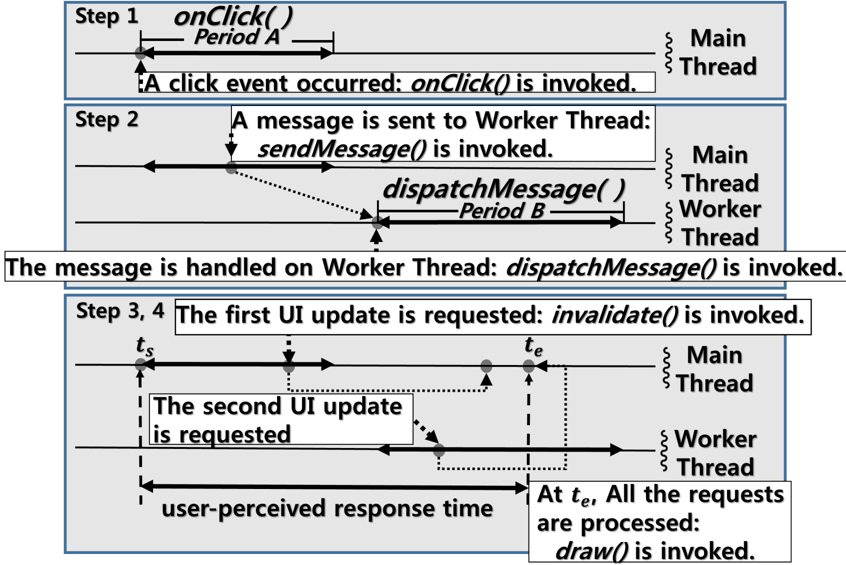


Figure 8: An example of identifying the user-perceived response time.

3.2.3 URA-based CPU Power Optimization Technique

Taking advantage of **ura**'s on-line identification of I^{oblv} for a given interactive session, we developed a new *cpufreq* governor, the `oninterval` *cpufreq* governor, for Linux CPU power management. Algorithm 1 describes how the `oninterval` *cpufreq* governor decides the CPU frequency. As with other Linux *cpufreq* governors, the CPU frequency is updated at each sampling period (e.g., 20 ms). The `oninterval` governor relies on endIdentifier of **ura** for keeping track of whether the current execution is in I^{perc} or I^{oblv}, as described on line 2 in Algorithm 1. Whenever the new interactive session S_i is started, the current execution interval type is set to I^{perc}. When endIdentifier detects the end of I^{perc}, it is changed to I^{oblv}. Based on this information, the `oninterval` governor employs the lowest

CPU frequency while executing I^{oblv}. Furthermore, the number of active cores is also restricted to one for further power reduction in I^{oblv}. Otherwise, decisions by the `ondemand` *cpufreq* governor [27], which is the default governor in most kernels for the Android Open Source Project, are applied in adjusting the CPU frequency. When the CPU utilization exceeds the predefined upper threshold (e.g., 95, U_{high} in Algorithm 1), for higher responsiveness, the `ondemand` *cpufreq* governor quickly switches to the maximum CPU frequency. On the other hand, if the CPU is less loaded (e.g., when the CPU utilization falls below 20, U_{low} in Algorithm 1), the governor gradually decreases the frequency. Therefore, when the new user input is initiated, the `oninterval` *cpufreq* governor can rapidly adapt to changing CPU utilizations.

3.3 Experimental Results

In order to evaluate the effectiveness of our proposed framework, we have implemented **ura** and **ura**-based CPU power management technique on the Samsung Exynos 4x12-based SMDK board running Android 4.0.4 (Ice Cream Sandwich). We modified the Dalvik VM interpreter for tracking all the method calls related to the identification of the end of the user-perceived response time. ModInterpreter and endIdentifier (which were described in Section 3.2.2) were implemented in the Dalvik VM and the Android Framework, respectively. We also modified the Linux kernel's *sysfs* interface slightly to support the **ura**-based CPU frequency governor. The `oninterval` *cpufreq* governor was also added to the Linux kernel, ver-

Algorithm 1 Pseudo code for the `oninterval` algorithm.

```
begin
    curExecIntervalType := getCurExecIntervalType()
    nextFreq := computeNextFreq_OnInterval(curExecIntervalType)
    setCpuFreq(nextFreq)
end

function CpuFreq computeNextFreq_OnInterval(curExecIntervalType)
begin
    if curExecIntervalType = I^perc then
        curFreq := getCurFreq()
        return computeNextFreq_OnDemand(curFreq)
    else if curExecIntervalType = I^oblv then
        //f_min is the minimum CPU frequency.
        return f_min
    endIf
end

function CpuFreq computeNextFreq_OnDemand(curFreq)
begin //The ondemand algorithm starts from this point.
    curUtil := getCpuUtilzation()
    if curUtil >U_high then
        //f_max is the maximum CPU frequency.
        return f_max
    else if curUtil <U_low then
        //f_min is the minimum CPU frequency.
        nextFreq := curFreq × 0.8
        return nextFreq
    else
        return curFreq
    endIf
end
```

sion 3.0.15. In our evaluations, we have experimented with 7 apps under different usage scenarios. Each app usage scenario consists of two consecutive interactive sessions. Table 1 summarizes selected apps and their usage

Table 1: Scenario descriptions of 7 benchmark apps.

App Name (Category)	Interactive Session ID	Interactive Session Description
News Republic (News)	S1	Launching
	S2	Viewing a list of all news
MocoSpace (Social Networking)	S3	Launching
	S4	Viewing a profile page
Photo Calendar (Photography)	S5	Launching
	S6	Selecting a photo album
Seesmic (Social Networking)	S7	Launching
	S8	Reading an article on Facebook
The Weather Network (Weather)	S9	Launching
	S10	Viewing a day into page
gReader (RSS Feed Reader)	S11	Launching
	S12	Reading a RSS feed
Web Browser (Web Browser)	S13	Launching
	S14	Clicking a link to an article

scenarios. An in-house scenario replay tool, which was implemented using the MonkeyRunner tool [28], is used to automatically execute the usage scenarios.

Prior to evaluating the efficiency of the proposed **ura**-based CPU power management framework, we first validated if **ura** can accurately estimate the user-perceived response time. In order to evaluate the accuracy of the estimated response time from **ura**, we have manually measured display-centric response times of our benchmark apps. For the manual measurement, we recorded the screen of the SMDK smartphone development board during the execution of each usage scenario with a digital video camera, which supports 30 fps frame rate. The recorded video was then analyzed frame by frame so that we can manually quantify the response time of the interactive sessions. Figure 9 compares user-perceived response times from man-

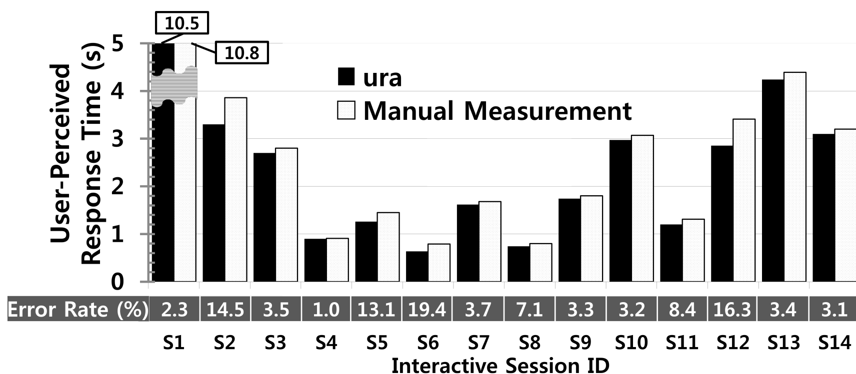


Figure 9: User-perceived response time differences between **ura** and manual measurements.

ual measurement and **ura**. The X-axis and the Y-axis denote various Android apps and their user-perceived response times, respectively. As shown in Figure 9, **ura** accurately estimates the user-perceived response times with an average error of 5.2% over manually measured times, thus achieving a sufficient accuracy for **ura**-based power/energy optimizations. Moreover, in our implementation on the smartphone development board, **ura** incurs additional computation overhead by up to only 1.2% of the user-perceived response time whenever the user-perceived response time is estimated.

Figure 10 shows the impact of the proposed `oninterval` governor on energy savings over varying durations of the think time for 14 interactive sessions. The result shows that the `oninterval` governor can save the CPU energy on average by 27.0% over the `ondemand` governor when the duration of the think time is 3 seconds. For S12 (gReader), the maximum energy saving of 65.6% is achieved with 5 seconds of the think time. For 9 out of 14 scenarios, the energy saving ratios increase as the duration

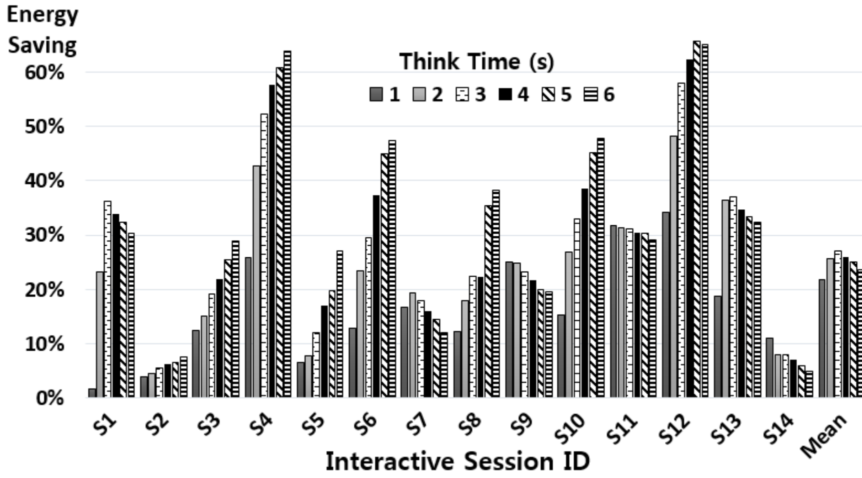


Figure 10: Changes in the average energy saving of 14 interactive sessions over varying durations of the think time.

of the think time grows. For S7, S9, S11, S13, and S14, on the other hand, the percentage of energy savings decrease as the duration of the think time grows after those intensive computation periods. This is because most background computations are completed within a couple of seconds (2, 1, 1, 3, and 1, respectively) after the end of the user-perceived response time. As shown in Figure 10, although the maximum energy savings were observed when the duration of the think time is 6 seconds, the `oninterval` governor can be useful even under shorter think times. When the think time is decreased to 1 and 2 seconds, our proposed `oninterval` governor can save the energy consumption, on average, by 21.7% and 25.7%, respectively over `ondemand`. Since the CPU power can account for up to 40% of the total power consumption in the latest smartphones [29], the `oninterval` governor can reduce the total power consumption by up to 10.3% when the

Table 2: A comparison of normalized user-perceived response times of seven second interactive sessions over varying durations of the think time in the first interactive sessions.

Interactive Session ID	Think Time (s)					
	1	2	3	4	5	6
S2	1.00	1.00	1.01	1.01	1.00	1.00
S4	1.00	1.01	0.99	1.01	1.00	0.99
S6	1.00	1.00	1.01	1.00	1.00	1.00
S8	0.99	0.99	0.99	1.01	0.99	0.99
S10	0.98	1.02	0.96	0.99	0.99	0.99
S12	1.01	0.99	1.00	1.01	0.99	0.99
S14	0.99	0.97	0.98	1.00	1.03	0.98

duration of the think time is 2 seconds.

In order to understand the impact of an aggressive DVS decision on the quality of user experience in the following interactive session, we compared, for each app usage scenario in Table 1, how the user-perceived response time of the second interactive session changes under the `oninterval` governor while varying the duration of the think time of the first interactive session. Table 2 shows normalized user-perceived response times of seven second interactive sessions, where user-perceived response times of seven second interactive sessions under the `ondemand` governor are used as baselines. We ran each scenario 100 times and the average of measured times was used for a comparison. For 5 out of 7 scenarios, normalized user-perceived response times range from 0.99 to 1.01. For S10 and S14, the `oninterval` governor increases the normalized user-perceived response time by up to 3% over the `ondemand` governor. Since our app usage scenarios all access remote servers through the wireless network (whose latency often fluctuates), we conclude that there is no significant difference in the user-perceived re-

sponse time of the second interactive sessions between the `ondemand` governor and `oninterval` governor.

Chapter 4

SmartDTM: Smart Thermal Management for Smartphones

4.1 Overview

Recently, smartphone users expect their devices to deliver PC-level performance. Therefore, in order to achieve the demanding requirement of the high performance, modern mobile processors have adopted multi-core architectures with increasing operating frequency. However, such efforts to build more-powerful mobile processors lead to an increase in on-chip power density, eventually resulting in high on-chip temperature. Since the highly elevated on-chip temperature negatively affects the reliability and the energy-efficiency of the devices, the effective thermal management is one of the crucial design requirements for the modern smartphones.

For the conventional computing systems such as PCs, there are many hardware-based thermal management solutions to cool down their processors. For example, a fan-cooled heat sink is the most commonly used method to reduce the on-chip temperature of the processors in the conventional computing systems [30]. However, although the hardware-based methods is a highly effective way to solving the thermal problems for the conventional systems, it is difficult to apply these solutions to the smartphones due to the

limited space of its form factor and the extra power consumption caused by these solutions [31]. Therefore, for the smartphones, the software-based thermal management methods are more appropriate to address the emerging thermal problems.

As a software-based thermal management method, the dynamic thermal management (DTM) [32] scheme is widely adopted by the smartphones. The main goal of the DTM scheme is to maintain the on-chip temperature below a *critical temperature*, $\texttt{Temp}_{critical}$, above which the processor chip could be damaged. For this purpose, in the DTM scheme, the temperature is periodically gathered from the on-chip thermal sensors. And then, when the on-chip temperature reaches a predefined *trigger temperature*, $\texttt{Temp}_{trigger}$, the maximum scaling frequency of the processor, which is set as the limit for the current dynamic voltage and frequency scaling (DVFS) policy, is lowered to control the over-heating of the chip. On the other hand, when the on-chip temperature drops under $\texttt{Temp}_{trigger}$, the maximum scaling frequency of the processor is gradually raised to ensure the optimal performance of the processor.

Although the DTM scheme can effectively mitigate the thermal problems by restricting the maximum scaling frequency of the processor to lower levels, as a side effect, degradation of user experience inevitably occurs while the DTM scheme is being applied. Furthermore, considering that the system performance level in I_S^{perc} is more likely to directly affect the quality of user experience (as discussed in Section 3.1), such DTM decisions should be carefully made, especially for when the current execution is in I_S^{perc}.

In this chapter, we propose a novel DTM technique for smartphones,

called SmartDTM, which ensures the quality of user experience while still avoiding the thermal violations. In SmartDTM, for each start point of every interactive session S_i, the length of $I_{S_i}^{perc}$ is predicted based on statistical analysis of the user-perceived response times for the past interactive sessions. Using the on-line prediction of the user-perceived response time, whenever the on-chip temperature reaches $\texttt{Temp}_{trigger}$ so that it is required to make a DTM decision, it is possible to know that the current execution is in $I_{S_i}^{perc}$ or $I_{S_i}^{oblv}$. And then, if it is identified that the current execution is in $I_{S_i}^{perc}$, we could carefully make the DTM decision for a higher quality of user experience. Moreover, by applying more aggressive dynamic voltage and frequency scaling (DVFS) techniques (e.g., the `oninterval` *cpufreq* governor in Section 3.2.3) to the thermal management, the on-chip temperature can be rapidly decreased during the execution of I_S^{oblv} without any negative effect on user experience.

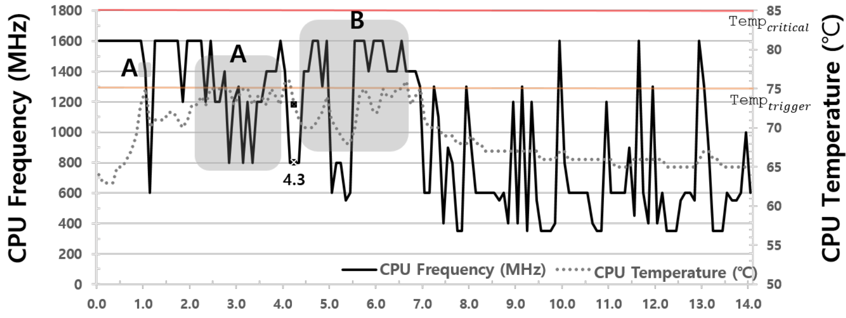
In order to evaluate our proposed technique, we implemented SmartDTM in the Android platform, version 4.4.2 (Kitkat) running on the Exynos 5410-based ODROID-XU+E board [33], which has current and voltage sensors to measure the power consumption of the on-board components including the big CPU cluster, the LITTLE CPU cluster, the GPU, and the DRAM module. Experimental results show that the proposed technique can improve the user-perceived performance by up to 37.96% over the Android's default DTM policy while maintaining the on-chip temperature of the processor below the critical temperature, which is set to 85 °C.

4.2 Motivation

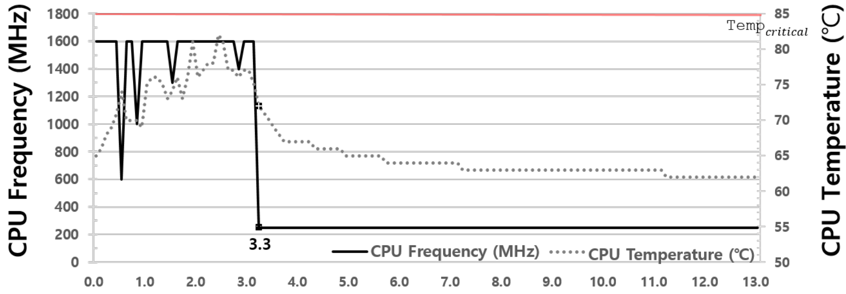
In order to better motivate our proposed DTM technique, we first illustrate how the user-perceived response time of an app launching interactive session S_L can be increased by the Android's default DTM policy using an example. Figure 11 (a) shows how the CPU frequency and on-chip temperature changes during the first 14.5 seconds after the **twitter** app is launched under the Android's default DTM policy. The X-axis, the Y-axis on the left side, and the Y-axis on the right side represent the elapsed time, CPU frequency, and CPU temperature, respectively. (Note that $\texttt{Temp}_{critical}$ is aggressively set to $85\,^{\circ}\mathrm{C}$ and $\texttt{Temp}_{trigger}$ is set to the default value, $75\,^{\circ}\mathrm{C}$.) In this example, at $t = 0$, the interactive session, S_L, is initiated by the **twitter** app launching. And then, since all the user-visible contents are fully drawn at $t = 4.3$, the user-perceived response time of S_L is identified as 4.3 seconds. When the CPU temperature reaches $75\,^{\circ}\mathrm{C}$, which is $\texttt{Temp}_{trigger}$ of the default DTM policy, at $t = 1.0$, the maximum scaling CPU frequency is changed from 1,600 MHz to 1,400 MHz in order to decrease the on-chip temperature. (Note that 1,600 MHz is the maximum operating CPU frequency the Exynos 5410 processor can run at.) After that, when the on-chip temperature is stabilized to below $75\,^{\circ}\mathrm{C}$, the default DTM policy immediately restores the maximum scaling CPU frequency to 1,600 MHz. Since the CPU temperature starts rising again at $t = 1.8$ and it reaches $75\,^{\circ}\mathrm{C}$ at $t = 2.4$, the maximum scaling CPU frequency is also lowered by the default DTM policy. In this time, the CPU temperature is barely maintained near $\texttt{Temp}_{trigger}$ instead of definitely dropping below $\texttt{Temp}_{trigger}$. Therefore,

the default DTM policy further restricts the maximum scaling frequency of CPU until it reaches 800 MHz, which leads to the cluster switching from big to LITTLE in order to avoid the thermal violation. Considering that the system performance level in $I_{S_L}^{perc}$ highly affects the quality of user experience, although the default DTM policy can achieve to effectively control the CPU temperature to $71.58\,°\mathrm{C}$, on average, it is obvious that such DTM decisions (as shown in the area A of Figure 11 (a)) may bring the user-perceived delay after all. In addition, taking into account that $\texttt{Temp}_{trigger}$ is set to much lower value than $\texttt{Temp}_{critical}$ in general, we can say that the default policy makes many unnecessary DTM decisions based on the pessimistic assumption of the current thermal trend.

In order to evaluate the user-perceived delay brought by the default policy, Figure 11 (b) also shows how the CPU frequency and on-chip temperature changes during the launching process of the **twitter** app under the no-DTM policy with the `oninterval` *cpufreq* governor. The X-axis, the Y-axis on the left side, and the Y-axis on the right side are same with those of Figure 11 (a). In this example, since the maximum scaling frequency of 1,600 MHz is not changed at all during the execution of $I_{S_L}^{perc}$, the user-perceived response time, 3.3 seconds, is shorter than that of the default DTM policy, 4.3 seconds (that is, the 30.3% degradation of the user-perceived performance). In addition, under the no-DTM policy, while the average and maximum CPU temperatures in $I_{S_L}^{perc}$, (which are $74.24\,°\mathrm{C}$ and $82\,°\mathrm{C}$, respectively), are higher than those of the default policy, we can observe that the CPU temperature still does not exceed $\texttt{Temp}_{critical}$. Moreover, although the CPU frequency is frequently increased to the maximum frequency of



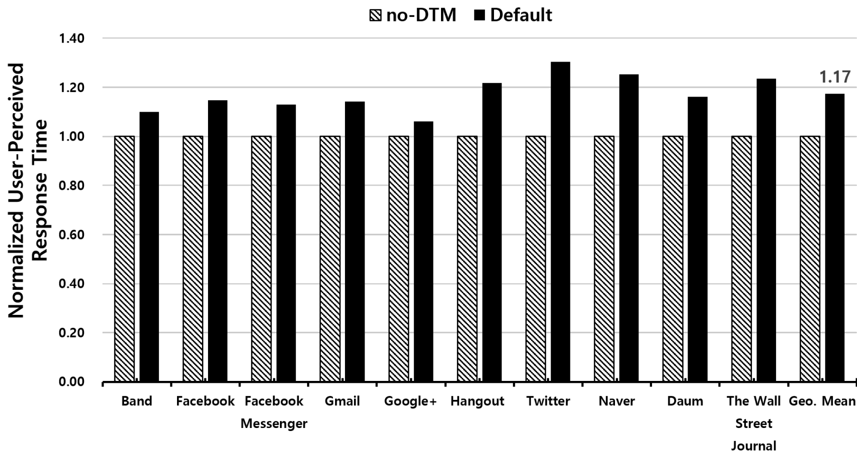
(a) Changes in the CPU frequency and temperature during the app launching session under the Android's default DTM policy.



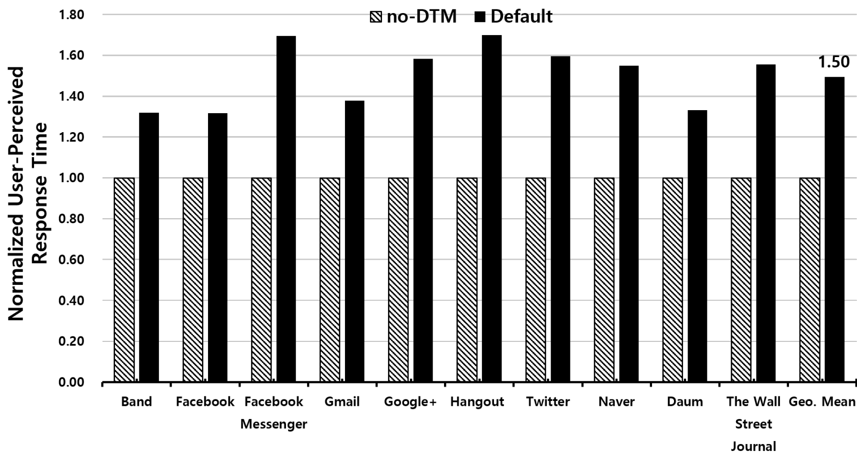
(b) Changes in the CPU frequency and temperature during the app launching session under the no-DTM policy with `oninterval` *cpufreq* governor.

Figure 11: An illustrative example of the difference between the Android's default DTM policy and the no-DTM policy with the `oninterval` *cpufreq* governor.

1,600 MHz even when executing in $I_{S_L}^{oblv}$ under the default DTM policy, thus increasing the CPU temperature by 76 °C at $t = 6.7$ (as shown in the area B of Figure 11 (a)), with the `oninterval` *cpufreq* governor, there is a considerable opportunity to decrease the CPU temperature right after the end of $I_{S_L}^{perc}$. In this case, if we knew when the $I_{S_L}^{perc}$ was going to end, and if we knew that the CPU temperature does not reaches $\texttt{Temp}_{critical}$ during the remaining execution of the $I_{S_L}^{perc}$, we could maintain the maximum scaling



(a) A comparison of normalized user-perceived response times between the no-DTM and default DTM policies when each interactive session is initiated at 65 °C.



(b) A comparison of normalized user-perceived response times between the no-DTM and default DTM policies when each interactive session is initiated at 70 °C.

Figure 12: Normalized user-perceived response time comparisons between the no-DTM and default DTM policies.

frequency as high as possible for a better quality of user experience.

In order to evaluate the possible improvement in the user-perceived performance of our proposed technique over different apps, we measured

user-perceived response times of launching interactive sessions under the no-DTM and default DTM policies. For better accuracy, each measurement was iterated for 30 times. Figure 12 (a) compares normalized user-perceived response times between the no-DTM and default polices for 10 launching interactive sessions. In this experiment, all of the launching interaction sessions are initiated at $65\,^{\circ}\mathrm{C}$, which is the base temperature of our ODROID-XU+E board in a normal state. The X-axis and the Y-axis denote various Android apps and their user-perceived response times, which are normalized to those of no-DTM. As shown in Figure 12 (a), even when the system is in the normal state, the default DTM policy can degrade the user-perceived performance by up to 30.0% and on average, 17.3% compared to the no-DTM policy. As the base temperature becomes close to $\texttt{Temp}_{trigger}$, the performance degradation further increases as well. When the base temperature is $70\,^{\circ}\mathrm{C}$, Figure 12 (b) shows that the user-perceived performance of the default policy can be decrease by 49.5%, on average, in comparison to the no-DTM policy.

4.3 Design and Implementation of SmartDTM

4.3.1 Basic Idea

In the default DTM policy, the DTM decisions to control the on-chip temperature of the processor are immediately made when the current temperature, $\texttt{Temp}_{current}$, reaches $\texttt{Temp}_{trigger}$. As a result, the maximum scaling CPU frequency, $\textsc{Max_Freq}_{scaling}$, is decreased from the maximum operating CPU frequency, $\textsc{Max_Freq}_{operating}$, which is 1,600 MHz in the

current system. Figure 13 illustrates how the default DTM policy controls the CPU temperature using $\text{MAX_FREQ}_{scaling}$ scaling. In order to quickly adapt to increasing the CPU temperature, $\texttt{Temp}_{current}$ is monitored at each sampling period (e.g., 100 ms). When $\texttt{Temp}_{current}$ exceeds $\texttt{Temp}_{trigger}$ is detected, the policy changes $\text{MAX_FREQ}_{scaling}$ to the one-step-lower level to reduce the CPU temperature. During this process, if the *cpufreq* governor is already working under the predefined lowest CPU frequency, MIN $_Freq_{scaling}$, because of the former DTM decisions, the policy does not further lower the CPU frequency below $\text{MIN_FREQ}_{scaling}$ to ensure the minimum acceptable quality of user experience. Otherwise, when the over-heat situation is successfully controlled by the policy, thus reducing $\texttt{Temp}_{current}$ below $\texttt{Temp}_{trigger}$, $\text{MAX_FREQ}_{scaling}$ is gradually raised until it reaches $\text{MAX_FREQ}_{operating}$.

The basic idea of the proposed DTM technique is that the DTM decisions should be carefully made when the current execution is in I_S^{perc} since the system performance level in I_S^{perc} significantly affects the quality of user experience. Figure 14 (a) illustrates an example of how the proposed scheme can avoid the DTM decision using on-line estimation of the user-perceived response time of each interactive session S_i and $t_{critical}$, which represents the time when the current temperature reaches $\texttt{Temp}_{critical}$. In order to identify that the current execution is in $I_{S_i}^{perc}$, t_{end}, which represents the end of $I_{S_i}^{perc}$, should be estimated first at t_{init}, which represents the beginning of S_i initiated by a certain user input. (We will describe how to estimate t_{end} in Section 4.3.3) Once t_{end} is estimated at t_{init}, it is possible to identify that the current execution is in $I_{S_i}^{perc}$ by calculating the time difference between t_{init}

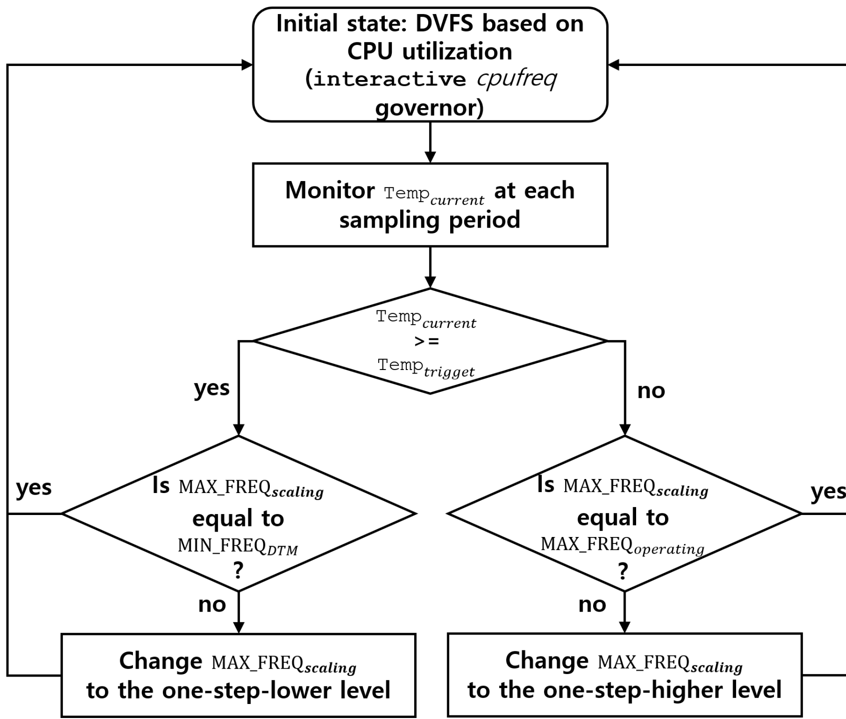


Figure 13: An overview of the default DTM policy with the `interactive` *cpufreq* governor.

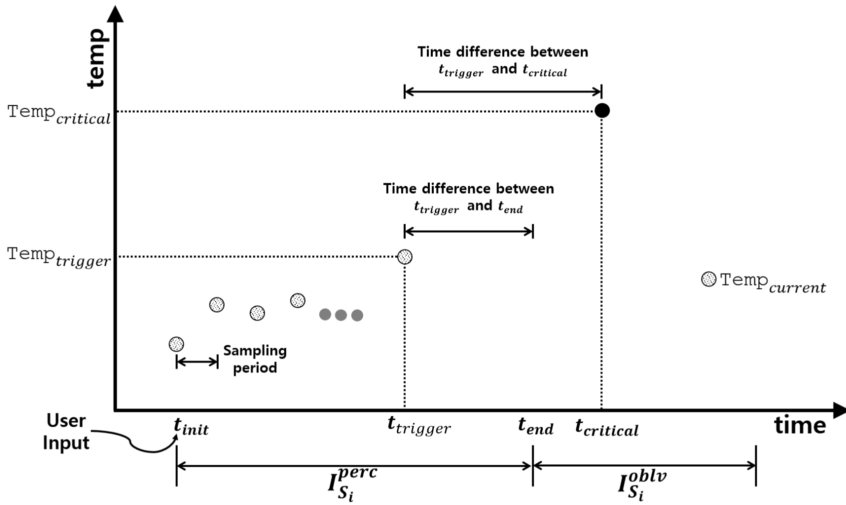
and t_{end} at every temperature-sampling point. When $\texttt{Temp}_{current}$ reaches $\texttt{Temp}_{trigger}$ at $t_{trigger}$, the proposed technique also should consider the MAX_FREQ$_{scaling}$ lowering. However, in SmartDTM, the time difference between $t_{trigger}$ and $t_{critical}$, which is the expected time when $\texttt{Temp}_{current}$ will reach $\texttt{Temp}_{critical}$, is calculated using *the worst-case temperature estimation model* before the DTM decision is made. (The worst-case temperature estimation model will be also explained in Section 4.3.4) Then, if the time difference between $t_{trigger}$ and $t_{critical}$ is longer than $t_{trigger}$ and t_{end}, MAX_FREQ$_{scaling}$ is not changed to ensure the quality of user experience

during the execution of $I_{S_i}^{perc}$. Furthermore, since the proposed SmartDTM relies on the `oninterval` *cpufreq* governor for the CPU frequency scaling, the lowest CPU frequency is employed when **ura** detects the end of $I_{S_i}^{perc}$. Therefore, the CPU temperature can be rapidly decreased while executing $I_{S_i}^{oblv}$.

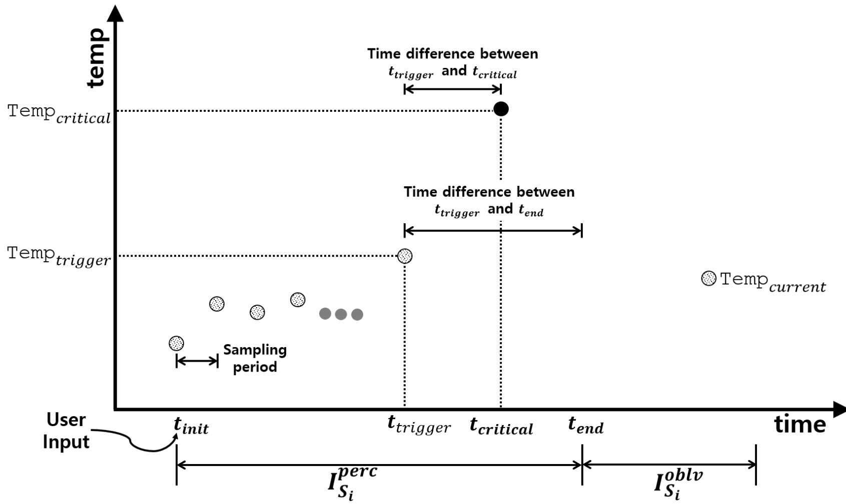
Although the quality of user experience is improved by restricting the DTM decisions during the execution of $I_{S_i}^{perc}$, avoiding the thermal violation is still a primary goal of the DTM policies. Figure 14 (b) illustrates the case when the proposed technique should lower $\text{MAX_FREQ}_{scaling}$ to reduce $\texttt{Temp}_{current}$. In this example, at $t_{trigger}$, the time difference between $t_{trigger}$ and $t_{critical}$ is much shorter than $t_{trigger}$ and t_{end}, which means that if we do not lower the CPU frequency then $\texttt{Temp}_{current}$ will soar above $\texttt{Temp}_{critical}$ at $t_{critical}$. In this case, DTM decisions by the default policy should be applied in reducing the CPU temperature.

4.3.2 Architectural Overview

Figure 15 shows an architectural overview of SmartDTM within the Android platform. SmartDTM consists of three main modules, **ura**, the thermal management module, and the `oninterval` *cpufreq* governor. As explained in Section 3.2.1, the main function of **ura** is to identify the end of $I_{S_i}^{perc}$. In order to support SmartDTM, two additional submodules, the interactive session classifier and the user-perceived response time predictor, are included in **ura**. When a user interacts with Android UI components such as BUTTON and TEXTVIEW WIDGET packages, the interactive session classifier makes the unique identifier for such interactive session using the UI



(a) The case when the DTM decision can be restrict during the execution of $I_{S_i}^{perc}$.



(b) The case when the DTM decision should be made to avoid the thermal violation during the execution of $I_{S_i}^{perc}$.

Figure 14: Examples of how the SmartDTM restricts the DTM decisions or not using on-line estimation of t_{end} and $t_{critical}$.

hierarchy of the Android VIEW system, and then classifies the interactive session according to the identifier. Whenever the end of $I_{S_i}^{perc}$ is identified

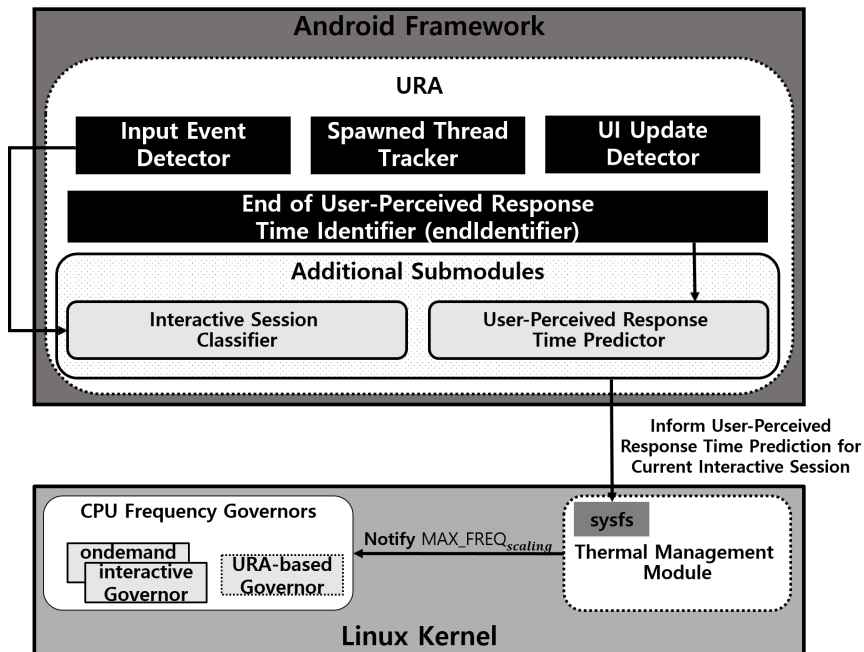


Figure 15: An architectural overview of SmartDTM.

by endIdentifier of **ura**, the user-perceived response time predictor accumulates the identification result about the user-perceived response time with the identifier, which is corresponding to the interactive session. Based on statistical analysis of the accumulated user-perceived response time information, the user-perceived response time of the current interactive session can be predicted. The thermal management module is responsible for applying the DTM decisions. In the thermal management module, $t_{critical}$ is estimated by using *the worst-case temperature estimation model* at each $t_{trigger}$. In order to avoid the thermal violation, the thermal management module verifies that $t_{critical}$ will not appear before the end of $I_{S_i}^{perc}$. In this case, the module does not change $\text{MAX_FREQ}_{scaling}$ in order to improve

the user-perceived performance. Otherwise, as with the default DTM policy, $\text{MAX_FREQ}_{scaling}$ is lowered by the thermal management module. If $\text{MAX_FREQ}_{scaling}$ is changed, the thermal management module notifies it to the `oninterval` *cpufreq* governor.

4.3.3 User-Perceived Response Time Prediction

In order to make a unique identifier to classify the interactive sessions, the interactive session classifier takes advantage of the Android's VIEW hierarchy system. Figure 16 represents an example of the GUI snapshot of the **twitter** app (on the top of Figure 16) and its VIEW hierarchy (on the bottom of Figure 16). As shown in this example, the GUI can be represented as a tree structure as well, where its terminal nodes correspond to the visible UI components. For example, each image button on the toolbar, each tweet post in the timeline, and each profile image (as shown in the area A, B, and C of the top of Figure 16) correspond to a ToolBarItemView node, a GroupedRowView node, and a ImageView node, respectively (as shown in the area A, B, and C of the bottom of Figure 16). Therefore, considering that an interactive session is initiated by a user input to a particular visible UI component such as the image button, it is possible to identify the interactive session by exploiting the information about the terminal node and its parents including the root as the interactive session identifier. In detail, whenever the input event detector of **ura** captures the events related to the user input, the classifier traverses the VIEW hierarchy from the terminal node, which handles the user input, to the root, DecorView, and makes a long string that is the concatenation of the names of all visited nodes. Since there can be

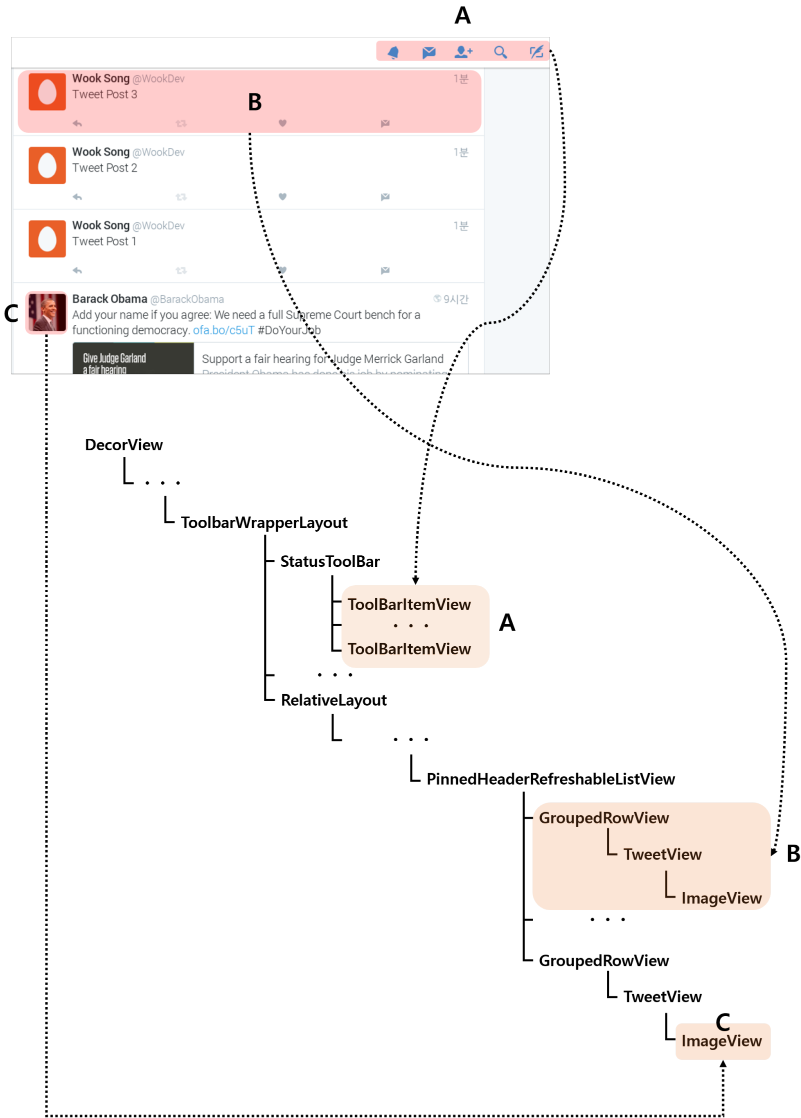


Figure 16: A GUI example of the **twitter** app and its VIEW hierarchy.

only one identifier for the terminal nodes having the same parent node, such interactive sessions, which is related to UI components having the identical identifier, are classified into a same group. In this example, the interactive

sessions initiated by the user inputs to the tweet posts in the timeline (as shown in the area B of the top of Figure 16) will be regarded as a single interactive session.

In order to make the prediction for the user-perceived response time of each classified interactive session, the past 30 user-perceived response times for each session are accumulated by the user-perceived response time predictor. Based on this collected information, an statistical analysis is performed for the prediction. In detail, since we assumed that the user-perceived response times of a specific interactive session are normally distributed, if we can know the mean and standard deviation of those user-perceived response times, then the specific percentile (e.g., 95.05% in the current implementation) of the distribution can be simply calculated as the expected user-perceived response time of the specific interactive session using the probit function.

4.3.4 Worst-Case Temperature Estimation Model

In order to build the worst-case temperature estimation model, it is necessary to measure the CPU power consumption under heavy usage scenarios since the CPU temperature is strongly related to the CPU power consumption. Figure 17 shows how the CPU power consumption and temperature of our Exynos 5410-based ODROID-XU+E board change under the heavy usage scenarios. In this measurement, since an app launching is one of the most performance heavy usage scenarios in general, we used 10 sequential interactive sessions of app launching, which are same interactive sessions used in Figure 12. The X-axis, the Y-axis on the left side, and the Y-axis

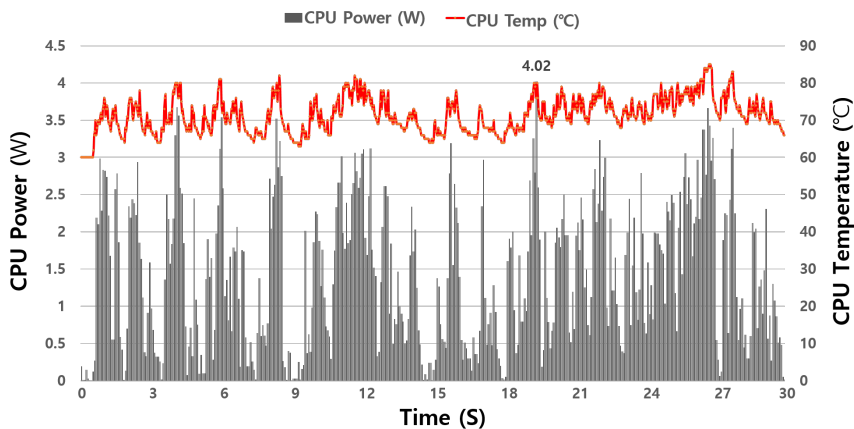


Figure 17: Changes in the CPU power consumption and temperature under heavy usage scenarios.

on the right side represent the elapsed time, CPU power consumption, and CPU temperature, respectively. As shown in Figure 17, the CPU power consumption varied dramatically depending on their workload and its maximum value was 4.02 W. Based on this observation, we safely assumed that the CPU power consumption of our ODROID-XU+E board can range from 4.00 W to 4.50 W under the performance heavy usage scenarios.

Although the CPU temperature is significantly influenced by the CPU power consumption, the workload characteristic is also an important factor affecting the CPU temperature. Therefore, in order to accurately build the worst-case temperature estimation model, it is required to observe the changes in the CPU temperature over varying the workload characteristics. For this observation, we compared the average CPU power consumption and the elapsed times corresponding to the CPU temperature changes from $t_{trigger}$ (i.e., $75\,^{\circ}\mathrm{C}$) to $t_{critical}$ (i.e., $85\,^{\circ}\mathrm{C}$) for each workload of the *mi-*

cro_bench binary, which is included in the Android Open Source Project. The detailed characteristics of the workloads is summarized in Table 3. Figure 18 shows the comparisons of the average CPU power consumption and the elapsed times between the various workload characteristics. The X-axis, the Y-axis on the left side, and the Y-axis on the right side denote various workloads, their average CPU power consumption, and the elapsed times, respectively. Since the CPU power consumption of two workloads, **memcpy_4** and **memset_4**, exceeds 5.50 W, which is extremely high for the target system, we excluded these workloads in this experiment. As shown in Figure 18, there are two workloads, **cpu_4** and **memread_3**, whose average power consumption ranges from 4.00 W to 4.50 W. Although **cpu_4** and **memread_3** have quite similar power consumption, which are 4.18 W and 4.34 W, respectively, the thermal characteristics of these workloads are completely different. In detail, while the CPU temperature has risen from 75 °C to 85 °C within 5.9 seconds during the execution of **cpu_4**, only 2.53 seconds have been taken for **memread_3**. For this reason, the **memread_3** workload is used to build the worst-case temperature estimation model using the linear-log regression.

4.4 Experimental Results

4.4.1 Experimental Environment

In order to evaluate the effectiveness of our proposed scheme, we have implemented the SmartDTM technique on the Exynos 5410-based ODROID-XU+E board running Android 4.4.2 (Kitkat). As extension modules of **ura**,

Table 3: Summary of the workloads in the *micro_bench* binary.

Benchmark Name	Description
CPU_#	Multi-threaded CPU intensive workload using # threads
MEMCPY_#	Multi-threaded cache intensive workload, which repeatedly calls the *memcpy* function using # threads
MEMCPY_COLD_#	Multi-threaded memory intensive workload, which repeatedly calls the *memcpy* function using # threads
MEMSET_#	Multi-threaded cache intensive workload, which repeatedly calls the *memset* function using # threads
MEMSET_COLD_#	Multi-threaded memory intensive workload, which repeatedly calls the *memset* function using # threads
MEMREAD_#	Multi-threaded cache intensive workload, which repeatedly reads values from an array using # threads
MEMREAD_COLD_#	Multi-threaded memory intensive workload, which repeatedly reads values from an array using # threads

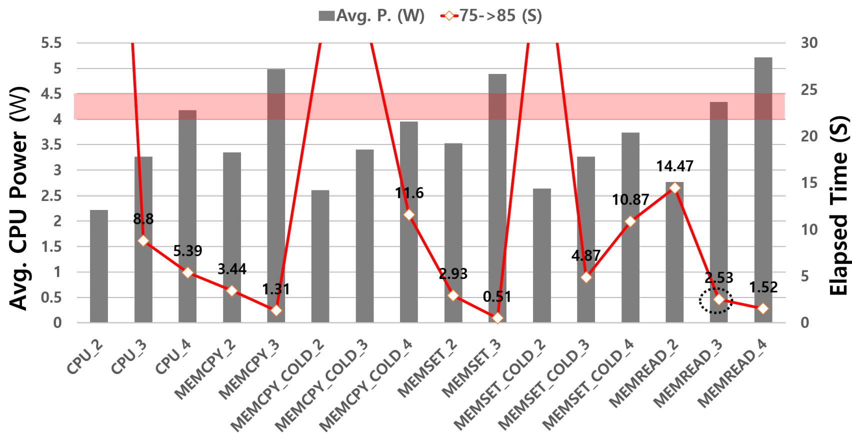


Figure 18: Differences of the average CPU power consumption and the elapsed times corresponding to the CPU temperature changes from 75 °C to 85 °C between the various workload characteristics.

the interactive session classifier and user-perceived response time predictor were implemented in the Android Framework. We added the thermal management module, which makes the DTM decisions based on the prediction

Table 4: Scenario descriptions of 10 apps used in the experiments.

App Name (Category)	**Interactive Session ID**	**Interactive Session Description**	**User-Perceived Response Time with no-DTM (s)**
Band (Social Networking)	S1	Launching	1.98
	S2	Viewing an article	0.53
Facebook (Social Networking)	S3	Launching	2.01
	S4	Clicking the search button	0.46
Facebook Messenger (Messenger)	S5	Launching	0.59
	S6	Changing tabs	0.24
Gmail (Mail)	S7	Launching	3.48
	S8	Reading a mail	3.42
Google+ (Social Networking)	S9	Launching	1.38
	S10	Viewing today's recommended featured collection	0.78
Hangout (Messenger)	S11	Launching	2.22
	S12	Opening a chat session	0.35
Twitter (Social Networking)	S13	Launching	3.35
	S14	Viewing a tweet post in the timeline	0.23
Naver (Web Portal)	S15	Launching	8.89
Daum (Web Portal)	S16	Launching	3.27
The Wall Street Journal (News)	S17	Launching	1.28

of the user-perceived response time, to the Linux kernel, version 3.4.5. In order to provide the user-perceived response time prediction for the thermal management module, a Linux kernel's *sysfs* file is also added to the Linux kernel. In our evaluations, we have experimented with 10 apps under different usage scenarios. Since our interactive session classifier exploits the Android's VIEW hierarchy system, it cannot support certain web-page-based-apps. Therefore, only launching usage scenario is used for those apps. Otherwise, each app usage scenario consists of two consecutive interactive sessions. Table 4 summarizes selected apps and their usage scenarios.

4.4.2 Performance Evaluation

Figure 19 shows the impact of the proposed SmartDTM technique on improvements in the user-perceived response times for 10 launching inter-

active sessions. In this experiment, each interactive sessions are initiated when the base temperature is $65\,^{\circ}\mathrm{C}$, which means the target system is in the normal state. In addition, since only launching interactive sessions can raise the CPU temperature above $\mathtt{Temp}_{trigger}$, $75\,^{\circ}\mathrm{C}$, from the base temperature, $65\,^{\circ}\mathrm{C}$, the launching interactive sessions among all the interactive sessions in Table 4 are selected for the evaluation. The result shows that our proposed DTM technique can improve the user-perceived response time on average 12.2% over the default DTM technique. For S13 (**twitter**), the proposed SmartDTM achieves the maximum improvement in the user-perceived response time of 21.3%. For 7 out of 10 scenarios, the user-perceived performance degradation was bounded by up to 3.0% in the SmartDTM technique. Even for the worst-case, S11 (**hangout**), the user-perceived response time is increased by 6.2% compared to the no-DTM policy, while the default DTM policy increases the user-perceived response time by 21.8%.

As the base temperature increases higher, more DTM decisions, which may degrade the user-perceived performance, are required to avoid the thermal violation during the execution of I_S^{perc}. In order to compare the degradation in the user-perceived performance, Figure 20 also shows the normalized user-perceived response times of 17 interactive sessions in Table 4 when the base temperature is $70\,^{\circ}\mathrm{C}$. As shown in Figure 20, it is observed that the user-perceived performance degradation of both the default and SmartDTM techniques is increased compared to the case when the base temperature is $65\,^{\circ}\mathrm{C}$. However, the proposed SmartDTM still can reduce the degradation by 21.5%, on average, over the default DTM technique. Moreover, for 9 of 17 interactive sessions, more than 50.0% of user-perceived response time

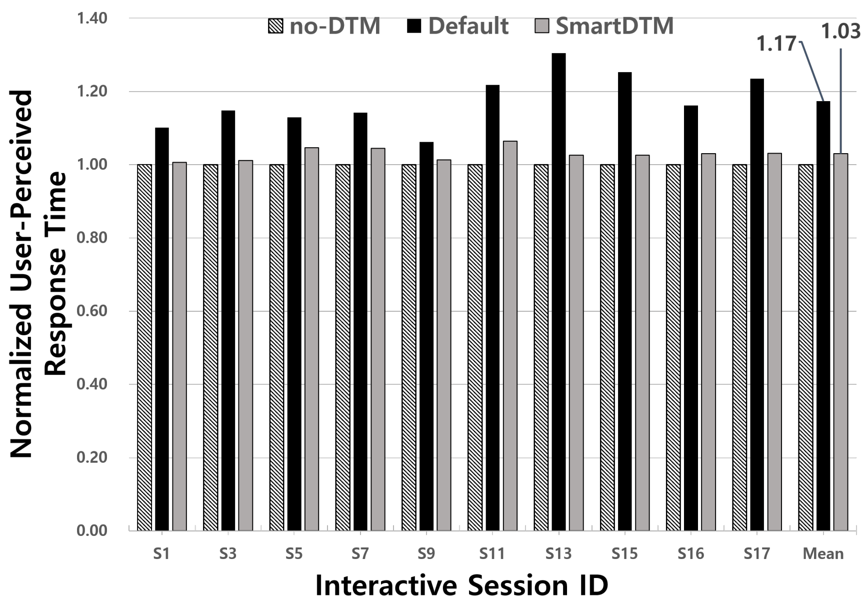


Figure 19: A comparison of normalized user-perceived response times for 10 launching interactive sessions when the base temperature is 65 °C.

delays are shown in the default DTM policy, while the proposed SmartDTM policy increases the user-perceived response times by more than 23.6% only for S11, S13, and S15. Otherwise, the user-perceived performance degradation was bounded by up to 18.1% in the SmartDTM technique.

4.4.3 Temperature Evaluation

In order to understand the impact of an aggressive DTM delaying decisions on the CPU temperature during the execution of the user-perceived response time interval, we compared how the CPU temperature changes under the default and SmartDTM policies while executing the user-perceived response time interval in Figure 21. The X-axis and Y-axis denote the in-

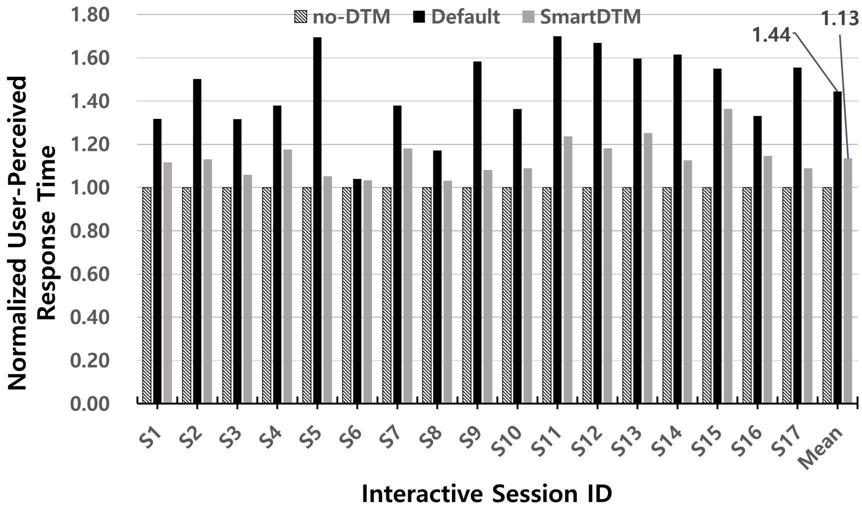


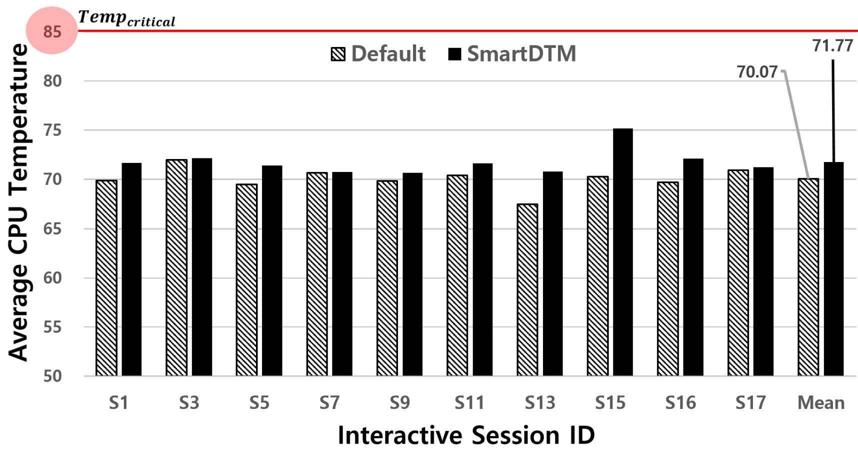
Figure 20: A comparison of normalized user-perceived response times for 17 interactive sessions when the base temperature is 70 °C.

teractive sessions and their average or maximum CPU temperatures while executing the user-perceived response time intervals, respectively. As shown in Figure 21 (a), when the base temperature is 65 °C, there are not noticeable differences of the CPU temperature between the default and proposed policies. The result shows that the SmartDTM scheme increases the CPU temperature by 1.70 °C on average. In addition, considering that the critical temperature of the target system is 85 °C, the maximum temperatures observed during the user-perceived response time interval under the SmartDTM could be acceptable results. As shown in Figure 21 (b), for S13 (a launching interactive session of **twitter**, the proposed technique increased the CPU temperature to 82 °C during the execution of the user-perceived response time.

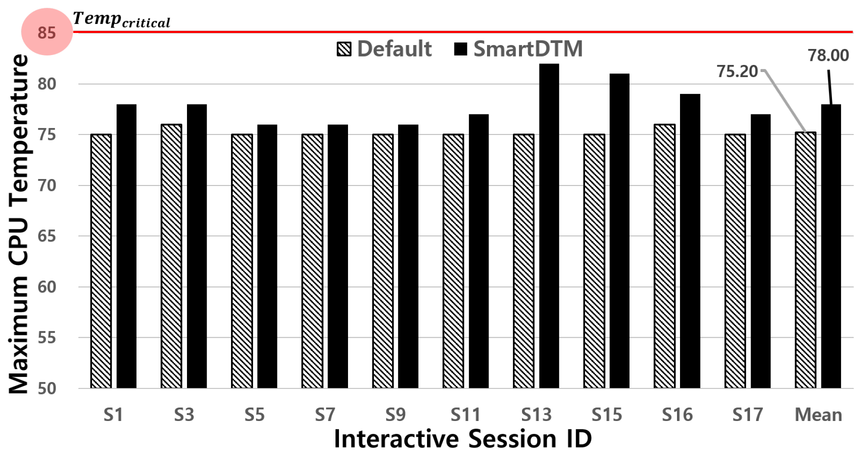
When the base temperature is changed close to the trigger temper-

ature of the target system, the CPU temperature during the execution of the user-perceived response time interval goes up. However, our proposed SmartDTM can avoid the thermal violence based on the worst-case temperature estimation model. In order to evaluate the effectiveness of the proposed SmartDTM on the thermal management, Figure 22 shows the differences of the average and maximum CPU temperature between the default and SmartDTM policies during the execution of the user-perceived response time interval. The X-axis and Y-axis are same as those of Figure 21. As shown in Figure 22 (a), the proposed technique also increases the CPU temperature by $2.09\,^{\circ}\mathrm{C}$ on average. In addition, although the much higher maximum CPU temperatures are shown in Figure 22 (b), we can see that the SmartDTM still can effectively avoid the thermal violence.

In order to evaluate the impact of the `oninterval` *cpufreq* governor on the CPU temperature dropping when the current execution is in the user-oblivious response time interval, we measured the length of the time interval between the time when the execution of the user-perceived response time interval ends and the time when the current temperature drops to $65\,^{\circ}\mathrm{C}$, the base temperature of the target system. The results are shown in Figure 23. The X-axis and Y-axis denote the interactive sessions and the elapsed times of them from the time when the execution of the user-perceived response time interval ends to the time when the current temperature drops to $65\,^{\circ}\mathrm{C}$. For the case when the base temperature is $65\,^{\circ}\mathrm{C}$, as shown in Figure 23 (a), the result shows that the proposed SmartDTM technique can reduce the length of the time interval between the time when the execution of the user-perceived response time interval ends and the time when the current temper-



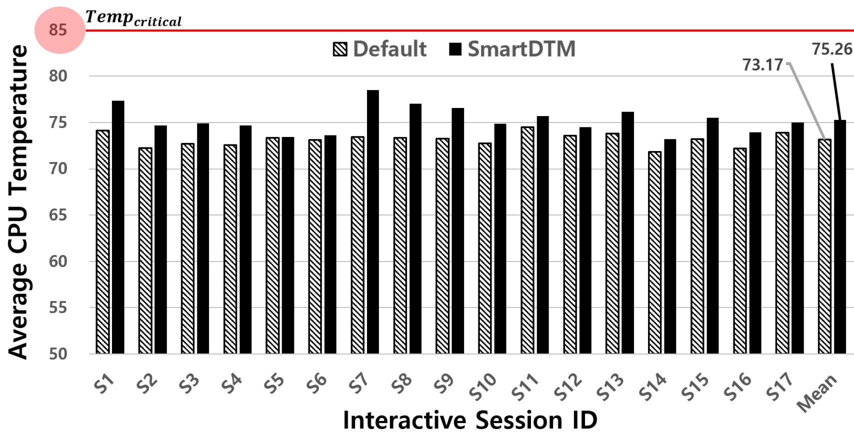
(a) A comparison of the average CPU temperature during the execution of the user-perceived response time interval.



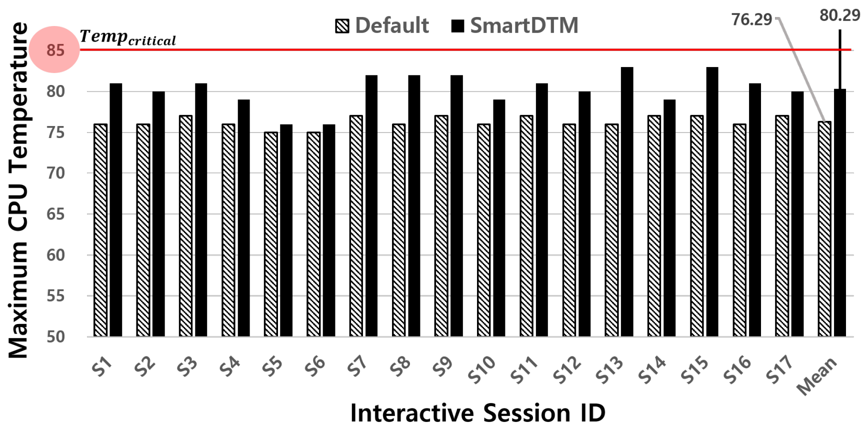
(b) A comparison of the maximum CPU temperature during the execution of the user-perceived response time interval.

Figure 21: The average and maximum CPU temperature differences between the SmartDTM and default policies when the base temperature is 65 °C.

ature drops to 65 °C by 58.34% due to the `oninterval` *cpufreq* governor. In addition, when the tasks are executed during the user-oblivious response time interval, it is observed that the default DTM policy cannot decrease



(a) A comparison of the average CPU temperature during the execution of the user-perceived response time interval.

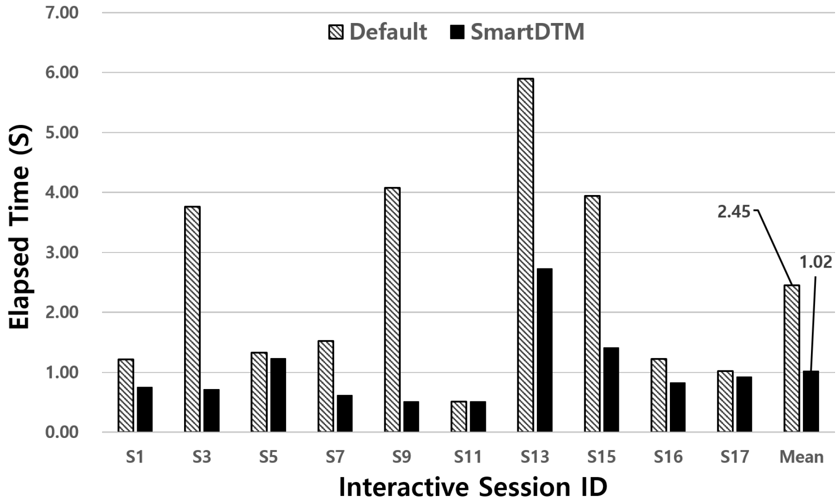


(b) A comparison of the maximum CPU temperature during the execution of the user-perceived response time interval.

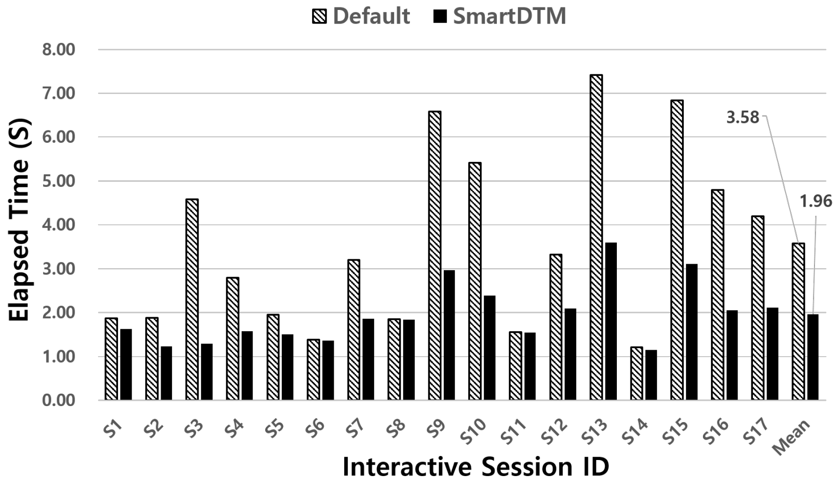
Figure 22: The average and maximum CPU temperature differences between the SmartDTM and default policies when the base temperature is 70 °C.

the CPU temperature even after the end of the user-perceived response time. As shown in Figure 23 (B), the proposed SmartDTM technique requires, on average, 45.25% less time to decrease the CPU temperature to the base

temperature during the execution of the user-oblivious response time interval. It is also shown that the proposed SmartDTM can be also effective in dropping the CPU temperature during the execution of the user-oblivious response time interval even when the base temperature is $65\,^{\circ}\mathrm{C}$. For S13, 7.42 seconds are required to decrease the CPU temperature to $65\,^{\circ}\mathrm{C}$, while the proposed SmartDTM only requires 3.60 seconds. In addition, for 10 of 17 interactive sessions, within 2.00 seconds, the proposed technique can drop the CPU temperature to the base temperature.



(a) The case when the base temperature is 65 °C.



(b) The case when the base temperature is 70 °C.

Figure 23: Distributions of the elapse times from the time when the execution of the user-perceived response time interval ends to the time when the current temperature drops to 65 °C.

Chapter 5

Personalized Optimization Framework

5.1 Motivation

In order to better understand how smartphone apps are used by different users, we collected detailed logs of smartphone usage from 21 college students and engineers living in Seoul. All the participants of this usage study were typical smartphone users, almost always carrying their smartphones with them around the clock.

For this usage study, we have developed a special Android app which collects various usage information in a non-intrusive fashion while users interact with their favored apps. This app automatically collects high-level information on smartphone use, including information about start and end of each app use, the detailed breakdown of how a user interact with each app, the list of processes maintained by the Linux per 30 minutes, and the on-off display status. In order to gather this information, our special app employs mechanisms for accessing system diagnostic event records, which are supported through the Android SDK. A local SQLite database is used for storing this collected information. We have distributed the special app to 37 study participants and 21 participants returned their logs.

From the analysis of the collected usage logs, we observed some distinct characteristics of app usage patterns, which formed the main motivation of our proposed *inter-app personalized optimization* approach. First, we have observed a well-known app usage tendency that only a small number of favored apps are heavily used. As shown in Figure 24, although a user had used on average 52 apps over the period of two weeks, only 10 apps had accounted for over 80% of total number of app uses. Second, we have also observed that there is a strong affinity on how related apps are used. In particular, we have observed that there are many pairs of apps that are used together in a particular situation. For example, Figure 25 illustrates such a strong affinity for *Memo App*. In order to understand the degree of app usage affinity, we conducted, for each app used, the per-app frequency as one of three apps launched right after *Memo App* was launched. Figure 25 shows that *Music Player App* was launched 27 times as one of three next apps launched following 38 *Memo App* launches. We have observed similar usage patterns from all 21 users. On average, 66% of apps were launched together with particular paring apps in more than half of their executions.

5.2 Design and Implementation of POA

5.2.1 Design Overview

The proposed POA framework consists of two main modules, the app usage modeling module and the app usage model-based optimization module. Figure 26 shows an overview of the POA framework within the Android platform. The app usage modeling module extracts meaningful app usage

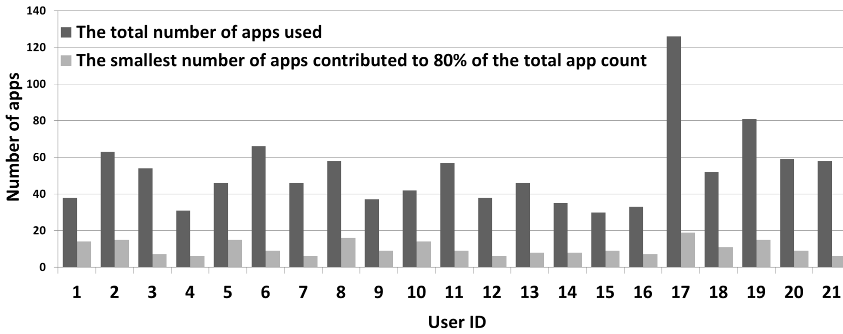


Figure 24: Per-user app usage distribution.

App	Music Player	Messenger	Settings	Battery Usage	Browser	MMS	Subway	Diary	News	Email	Control Panel	Calendar
Count	27	12	4	4	4	4	3	2	2	2	1	1

Figure 25: Per-app frequency as one of three next launched apps right after each *Memo App* launch (Total 38 *Memo App* launches).

patterns from execution logs and the app usage model-based optimization module is responsible for exploiting AUMs for various optimizations. In order to efficiently evaluate various optimization schemes developed within the POA, we have built several off-line custom POA support tools as well. The logger, which is explained in Section 5.1, and analyzer tools are used for collecting app usage logs and analyzing them off-line, respectively. The log replayer tool is used to quickly execute the app usage sequence extracted from the real app usage logs collected by limiting the time spent in each app usage to 10 seconds so that different solutions can be quickly explored.

Figure 27 illustrates how the POA framework extracts app usage patterns. In this example, it is clear that the user has a tendency to launch both *Subway App* and *Music Player App* at similar times. Furthermore, we can observe that the user launches *Subway App* prior to *Music Player App*. Based

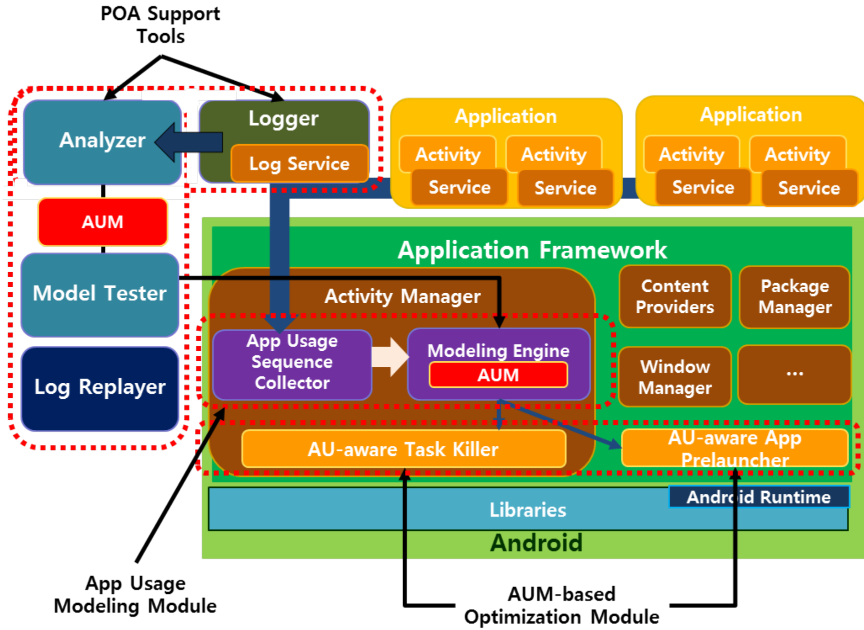


Figure 26: An architectural overview of the proposed POA framework.

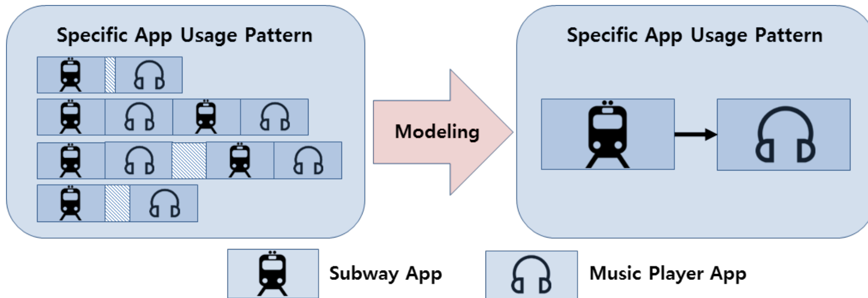


Figure 27: An example of building an app usage model.

on this app usage log information, the app usage modeling module of the POA framework can estimate this particular usage tendency between *Subway App* and *Music Player App* as shown in Figure 27.

Once the AUM is constructed from the app usage modeling module,

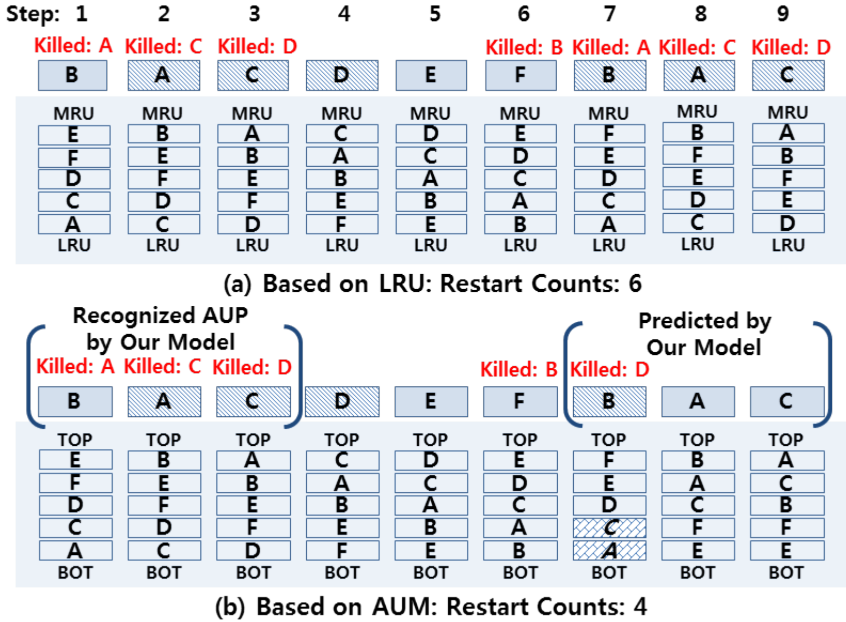


Figure 28: An example of using an app usage pattern in optimizing the LRU-based task killing policy.

we can take advantage of the AUM's knowledge on the user behavior for improving the system performance and user experience. Figure 28 conceptually illustrates how the AUM can be used in improving the LRU-based task killing policy of the Android platform. In this example, we assume that the LRU stack contains only 5 apps for a simplicity[1]. We further assume that the sequence of three apps, $B \rightarrow A \rightarrow C$, is frequently launched. Under the Android's default LRU-based task killing policy, both A and C would be killed at the steps 7 and 8 even though they will be reused right after B is launched. With our proposed AUM, however, this $B \rightarrow A \rightarrow C$ pattern can be recognized, and the AUM can predict that both A and C will be used

[1]In the Android platform, the depth of the LRU stack is 15 by default.

right after *B* is used. Therefore, *D* will be killed instead at the step 7. For this example app sequence, the AUM-based killing policy requires four app restarts. On the other hand, the LRU-based policy requires six app restarts.

5.2.2 App Usage Modeling Module

The app usage modeling module is implemented as an additional module of *ActivityManager* of the Android platform. It consists of two submodules, the app usage sequence collector and modeling engine. The app usage sequence collector accumulates past app usage data, which are used for modeling the user's app usage patterns. When app usage data are collected, we also collect various system-related profile information such as the screen on-off state and available memory capacity. App usage data are collected when *ActivityManager* receives a request to launch an app in the *startActivityLocked* method in *ActivityStack*. When the LRU stack of running processes is checked to manage the total number of running apps in the *updateOomAdjLocked()* method of *ActivityManagerService*, system-related profile information are collected. The modeling engine builds a user-specific AUM dynamically based on the information collected by the app usage sequence collector. As an independent module, it creates and initializes a AUM module at the boot time. Then, user's app usage data collected by the app usage sequence collector are passed to the modeling engine.

5.2.3 Usage Model-Based Optimization Module

The usage model-based optimization module consists of various optimization-specific submodules. In the current implementation, two optimization submodules exist, the app usage (AU)-aware task killer and app usage (AU)-aware app prelauncher (which will be described in details in Chapter 4.) Submodules apply the user-specific AUM to improve the performance and the user experience. When the optimization modules apply their policies, they request hints on future app executions from the app usage modeling module.

5.3 App Usage Model Construction

We have developed two heuristics for building an app usage model from collected execution logs. Before explaining two heuristics, we first define the following terms and notations that are useful in describing our heuristic. Let a sequence $S = \langle a_1, a_2, \ldots, a_l \rangle$ represent an app usage log where a_i indicates the i-th launched app. We assume that a_1 is the first app launched while a_l indicates the last app launched. We also define a set $\mathbb{D}_S$ of distinct apps in the app usage log S as $\mathbb{D}_S = \{a_{i_1}, \ldots, a_{i_k}\}$ where the set $\mathbb{D}_S$ consists of distinct apps launched in S. (That is, for all $p \neq q, a_{i_p} \neq a_{i_q}$.) For example, $\mathbb{D}_{\langle x,y,y,x \rangle} = \{x, y\}$ if $S = \langle a_1, a_2, a_3, a_4 \rangle = \langle x, y, y, x \rangle$.

5.3.1 P-AUM: Pattern-based App Usage Model

The basic idea of the pattern-based app usage model (P-AUM) is that frequently occurring usage patterns of the past are more likely to appear in

the future. In order to build a P-AUM heuristic, a past app usage log S is maintained. When the model is asked to decide apps which are likely to be launched next, the n most recently launched apps are first obtained from the past app usage log S. We call these n most recently launched apps as the current pattern. Then, the P-AUM heuristic finds candidate positions from the app usage log $S = \langle a_1, a_2, \ldots, a_l \rangle$. Candidate positions in the sequence represent where similar usage patterns as the current pattern are likely to be found. The candidate position, called a similar position, is selected by using the *Damerau-Levenshtein distance* algorithm [34]. We represent a similar position by its index in S.

In this algorithm, a similarity metric, called the *edit distance*, is calculated for measuring how similar two strings are. The *edit distance* represents the minimum number of edits to transform one string into the other by insertion, deletion, substitution and transposition. In the P-AUM heuristic, an app launch is modeled as a character of a string. The P-AUM heuristic can predict a tendency of the usage pattern in the past even though a current app sequence is a little different from past sequences by calculating the *edit distance*. In addition, this algorithm is also allowed to transpose two adjacent characters. Therefore, it can also find similar positions when two apps are executed in the reverse order.

Figure 29(a) illustrates the modeling process explained above. The P-AUM heuristic finds similar positions with the current pattern X - Y - Z and groups similar positions into the sets $\mathbb{M}_1, \mathbb{M}_2$ and $\mathbb{M}_3$ based on the *edit distance*. The P-AUM heuristic picks the set $\mathbb{M}$ with the minimum *edit distance* for estimating each app's immediacy on future launch given the current app

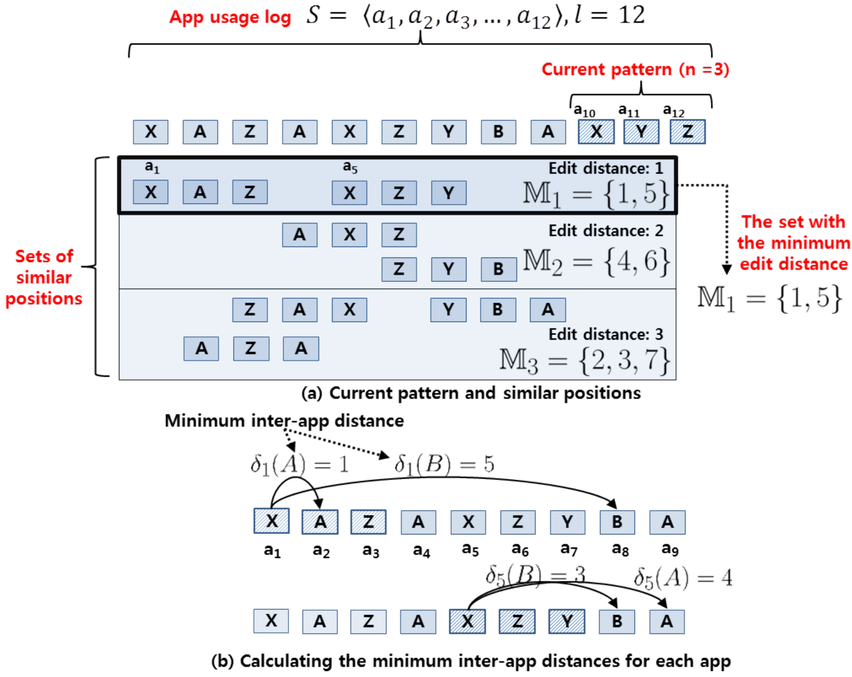


Figure 29: An example of how the P-AUM heuristic finds apps to be launched next.

sequence. For example, in Figure 29(a), the P-AUM heuristic selects $\mathbb{M}_1$.

Once the set $\mathbb{M} = \{s_1, \ldots, s_k\}$ with the minimum edit distance is decided, for each app x, we compute the average inter-app distance called as $pscore$. Intuitively, $pscore$ of an app x indicates how soon x will be likely to be launched again. The lower the $pscore$ of x, the sooner the launch of x. In order to compute $pscore$ of x, we first compute the minimum inter-app distance $\delta_{s_i}(x)$ for a given $S = \langle a_1, \ldots, a_l \rangle$ and a similar position $s_i \in \mathbb{M}$ as follows:

Definition 1 (Minimum Inter-App Distance)**.**

The minimum inter-app distance $\delta_{s_i}(x)$ of an app x given a similar

position s_i is defined as k iff

(i) $1 \le s_i \le j \le l$ *and* $x = a_j$,

(ii) There is no t such that $a_t = x$ for $s_i < t < j$, and

(iii) $k = |\mathbb{X}| - 1$ *where* $\mathbb{X} = \mathbb{D}_{\langle a_{s_i}, a_{s_{i+1}}, \ldots, a_{s_j} \rangle}$

If $\delta_{s_i}(x)$ is not defined (i.e., when the conditions (i) or (ii) is not satisfied), $\delta_{s_i}(x)$ is defined as $|\mathbb{D}_S|$.

For example, in Figure 29(b), from the similar position 1 (i.e., X - A - Z), the minimum inter-app distance of A, $\delta_1(A)$, is 1 and that of B, $\delta_1(B)$, is 5. Although not shown in Figure 29(b), $\delta_1(X)$, $\delta_1(Y)$ and $\delta_1(Z)$ are calculated similarly. The P-AUM heuristic repeats the same procedure from each similar position in $\mathbb{M}$.

Once $\delta_{s_i}(x)$ for all $i \in \mathbb{M}$ is computed, we compute $pscore(x, \mathbb{M})$ of an app x as follows.

$$pscore(x, \mathbb{M}) = \frac{\sum_{i \in \mathbb{M}} \delta_i(x)}{|\mathbb{M}|}$$

Based on the $pscore$ values, the P-AUM heuristic decides the relative order of future app launches. An app, which is assigned to a lower $pscore$, is likely to be launched in a near future.

The time complexity of building a P-AUM model for a given $S = \langle a_1, a_2, \ldots, a_l \rangle$ and the current pattern of the length n can be estimated as follows:

(i) Computing the *edit distance* of all positions to decide the set $\mathbb{M}$ of similar positions with the minimum *edit distance*: $O(n^2 \cdot l)$. This

is because the complexity of the *Damerau-Levenshtein distance* algorithm is $O(|M| \cdot |N|)$ for given two strings M and N. The P-AUM heuristic computes the edit distances between the current pattern of a length n and a past pattern of a length n from a_1 to a_{l-n+1}. (i.e., $|M| = |N| = n$)

(ii) Computing the $pscore$ for all apps in $\mathbb{D}_S$: $O(|\mathbb{D}_S| \cdot |\mathbb{M}| \cdot l)$

Although, in theory, the time complexity depends on n, $|\mathbb{D}_S|$, $|\mathbb{M}|$, and l, the actual computation time in building a P-AUM model is dominated by l. This is because the app sequence length l is much bigger than n, $|\mathbb{M}|$ and $|\mathbb{D}_S|$. For example, in real app usage logs, when the maximum app sequence length l is 2500, $|\mathbb{D}_S|$ is 50 and the average value of $|\mathbb{M}|$ is 14.82. In the current implementation, we use 4 for n. In our implementation on the Nexus S with real app usage logs, it took on average 20.24 ms in running the P-AUM heuristic whenever a new app was launched for the given $S = \{a_1, \ldots, a_{2500}\}$ and $|\mathbb{D}_S| = 50$.

5.3.2 C-AUM: Clustering-based App Usage Model

The clustering-based app usage model (C-AUM) is motivated by an observation that several related apps are often launched together in a particular situation. For example, if a user likes to listen to music while browsing web pages, *Browser App* is likely to be strongly related with *Music Player App*. Based on this observation, we developed an app usage model which clusters strongly related apps based on a metric that characterizes the launch affinity among apps.

In order to represent the launch affinity between two apps, we first compute the launch radius $r_i(x)$ of an app x relative to the app a_i in $S = \langle a_1, \ldots, a_i, \ldots, a_l \rangle$ as follow:

Definition 2 (Launch Radius)**.**

The launch radius $r_i(x)$ of an app x relative to a_i in S is defined as k iff

(i) $x = a_j$ *in* S

(ii) $k = |i - j|$ *and*

(iii) *There is no t such that $a_t = x$ for $|i - t| < k$.*

Intuitively, if $r_i(x)$ is small, the app x is likely to have a high launch affinity with the app a_i. Given $S = \langle a_1, \ldots, a_l \rangle$, the same app x can appear in multiple locations. For example, the app x may launch as a_1, a_3, a_5 and a_{100}. Therefore, we compute the clustering affinity $ca(x, y)$ of the apps x and y by combining the average launch radius of y relative to x and the average launch radius of x relative to y. The average launch radius $\overline{r(y|x)}$ of y relative to x is computed as follows:

$$\overline{r(y|x)} = \frac{\sum_{i \in S_x} (l - r_i(y))^2}{|S_x|}$$ where $S = \langle a_1, \ldots, a_l \rangle$ and $S_x = \{j \in \{1, \ldots, l\} | a_j = x \text{ in } S\}$.

If the apps x, y are closely related in their launch orders, $\overline{r(y|x)}$ will be large because $r_i(y)$'s will be small values. The average launch radius of x relative to y, $\overline{r(x|y)}$ is similarly computed. Finally, the clustering affinity $ca(x, y)$ of the apps x and y are defined as follows:

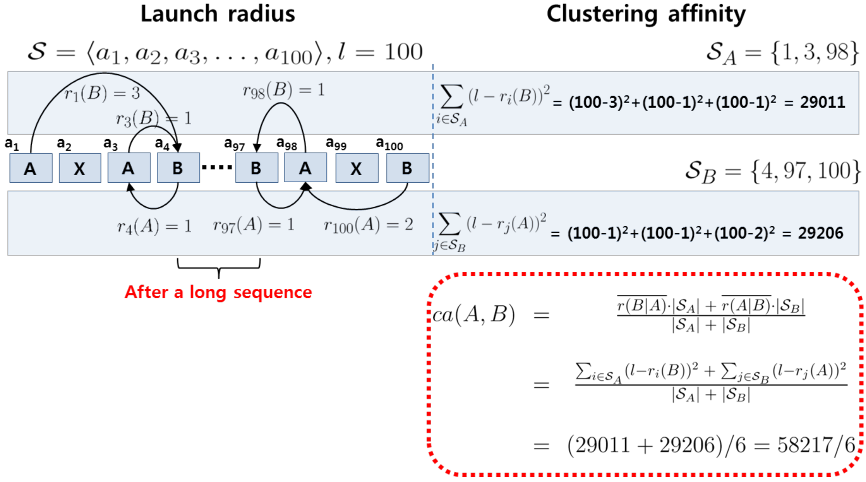


Figure 30: An example of the launch radius and the clustering affinity between two apps.

Definition 3 (Clustering Affinity)**.**

The clustering affinity $ca(x, y)$ between two apps x and y is defined as follows:

$$ca(x, y) = \frac{\overline{r(y|x)} \cdot |S_x| + \overline{r(x|y)} \cdot |S_y|}{|S_x| + |S_y|}$$

If $ca(x, y)$ is not defined, $ca(x, y)$ is assumed to be a negative value.

For example, in Figure 30, the launch radius from the first A to B, $r_1(B)$, is 3, and from the second A to B, $r_3(B)$, is 1. In this manner, the C-AUM heuristic calculates all the launch radiuses between A and B, and then, determines the clustering affinity between A and B by combining the average launch radiuses. In order to give more weights on smaller launch radiuses, the average launch radius is computed using the squared value, $(l - r_i(y))^2$. In the example in Figure 30, $ca(A, B)$ is given by $(((100 - 3)^2 + (100 - 1)^2 + (100 - 1)^2) + ((100 - 1)^2 + (100 - 1)^2 + (100 - 1)^2 +$

$(100 - 2)^2))/6$. The higher the clustering affinity, the stronger the relation of related apps is.

After all the cluster affinity values are computed over all (X, Y) pairs in $X, Y \in \mathbb{D}_S$, the C-AUM heuristic clusters apps using the *single-linkage clustering* algorithm [35]. Figure 31 shows an example of how the C-AUM heuristic builds a model by clustering apps using the *single-linkage clustering* algorithm[2]. Starting from an initial setting where each app is considered as a separate cluster, the *single-linkage clustering* algorithm progressively constructs all meaningful clusters as follows:

Step 1: *C* and *D*, which are the most closely related app pair ($ca(C, D)$ is 19^2), are clustered as *Cluster 1*.

Step 2: *A* and *B*, which are the next most closely related app pair ($ca(A, B)$ is 18^2), are clustered as *Cluster 2*.

Step 3: Because *D* (in *Cluster 1*) and *E* has the next largest cluster affinity value, *Cluster 1* and *E* are clustered as *Cluster 3*.

Step 4: Because $ca(B, C)$ is the next largest, *Clusters 2* and *3* becomes *Cluster 4*.

Since clusters are identified in the order of decreasing cluster affinity values, earlier identified clusters in the above algorithm are regarded as more strongly related apps over later identified clusters. For example, in Figure 31, *Cluster 1* has more strongly related apps than *Cluster 3*. The C-AUM heuristic represents how an app x is related to a given set of apps using the $cscore$ metric. Prior to explaining how to compute $cscore$ of an

[2]In this paper, we do not include a detailed description of the *single-linkage clustering* algorithm, which can be found in [35].

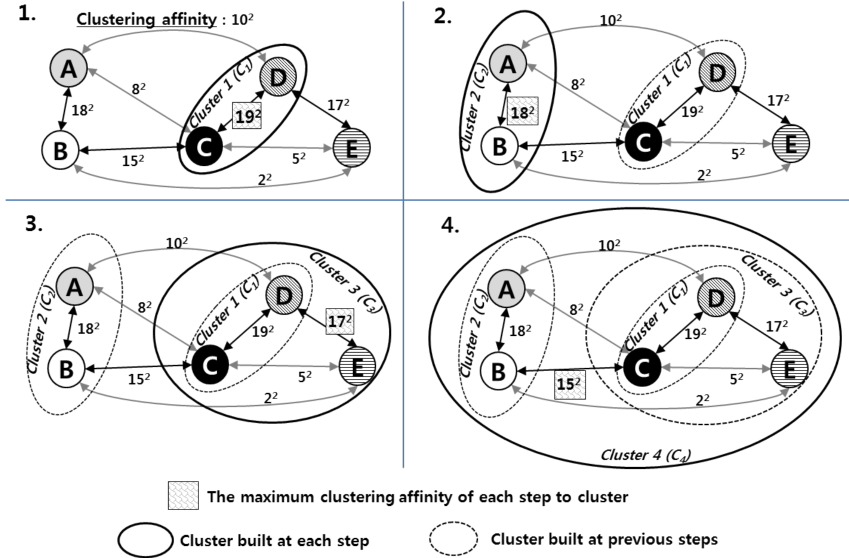


Figure 31: An example of building clusters using the *single-linkage clustering* algorithm.

app, we first introduce some related concepts needed in computing $cscore$ values. Let a sequence $C = \langle c_1, c_2, \ldots, c_z \rangle$ represent a cluster-construction sequence of where c_i indicates the i-th built cluster. We assume that c_1 is the first built cluster while c_z indicates the last built cluster. In addition, let a set $\mathbb{E}_{c_i} = \{a_{i_1}, \ldots, a_{i_k}\}$ represent distinct app entries which are contained in the cluster c_i. For example, in Figure 31, $C = \langle c_1, c_2, c_3, c_4 \rangle$ and $\mathbb{E}_{c_3} = \{C, D, E\}$.

We also define three support functions, the start cluster $\alpha(x)$, the parent cluster $\rho(c)$, and the strongest cluster $\sigma(\mathbb{C})$ for a given $C = \langle c_1, \ldots, c_z \rangle$ as follows:

Definition 4 (Start Cluster)**.**

The start cluster $\alpha(x)$ *of an app* x *is defined as* c_k *iff*

(i) $x \in \mathbb{E}_{c_k}$ *and*

(ii) There is no t *such that* $|\mathbb{E}_{c_t}| < |\mathbb{E}_{c_k}|$ *for* $x \in \mathbb{E}_{c_t}$*.*

For example, in Figure 31, the start cluster of the *App A*, $\alpha(A)$, is c_2 (*Cluster 2*).

Definition 5 (Parent Cluster)**.**

The parent cluster $\rho(c)$ *of a cluster* c *is defined as* c_k *iff*

(i) $\mathbb{E}_c \subset \mathbb{E}_{c_k}$ *and*

(ii) There is no t *such that* $|\mathbb{E}_{c_t}| < |\mathbb{E}_{c_k}|$ *and* $\mathbb{E}_c \subset \mathbb{E}_{c_t}$*.*

For example, in Figure 31, the parent cluster of c_2, $\rho(c_2)$, is c_4.

Definition 6 (Strongest Cluster)**.**

The strongest cluster $\sigma(\mathbb{C})$ *of a given set* $\mathbb{C}$ *of clusters identified from the cluster-construction sequence* $C = \langle c_1, \ldots, c_z \rangle$ *is defined as* c_k *iff*

(i) $c_k \in \mathbb{C}$ *and*

(ii) There is no t *such that* $c_t \in \mathbb{C}$ *and* c_t *appears earlier than* c_k *in* C*.*

For example, the strongest cluster $\sigma(\mathbb{C} = \{c_2, c_3, c_4\})$ is c_2.

Using these support functions, Algorithm 2 describes how the $cscore(x)$ of each app x is computed for a given cluster-construction sequence of clusters $C = \langle c_1, \ldots, c_z \rangle$ and a given set $\mathbb{A} = \{a_1, \ldots, a_n\}$ of apps. (In order

Algorithm 2 Computing the $cscore$ of an app x (given a set $\mathbb{A}$ of apps)

```
for each app x in 𝔸 do
    ℂ ← ℂ ∪ {α(x)}
    cscore(x) ← 0
end for
score ← 1
while ℂ ≠ {c_z} do
    c ← σ(ℂ)
    for each app x in 𝔼_c do
        if cscore(x) is still not assigned then
            cscore(x) ← score
        end if
    end for
    for each cluster c' in ℂ do
        if c' ≠ c_z then
            if ρ(c') = ρ(c) then
                ℂ ← ℂ − {c'}
            end if
        end if
    end for
    ℂ ← ℂ ∪ {ρ(c)}
    score ← score + 1
end while
```

to explore which apps are strongly related with the current sequence of app launches, the C-AUM heuristic selects $\mathbb{A}$ as the most recently launched n apps. That is, for a given $S = \langle a_1, \ldots, a_l \rangle$, $\mathbb{A} = \{a_l, a_{l-1}, \ldots, a_{l-n+1}\}$.)

For example, shown as Step 4 in Figure 31, if *Apps A* and *E* are selected as $\mathbb{A} = \{A, E\}$, Algorithm 2 works as follows:

Step 1: $\mathbb{C} = \{\alpha(A), \alpha(E)\} = \{c_2, c_3\}$ and $cscore(A) = cscore(E) = 0$ at the initial state.

Step 2: $cscore(B) = 1$ by $\sigma(\mathbb{C}) = c_2$ and $\mathbb{C}$ changes to $\{c_3, c_4\}$

Step 3: $cscore(C) = cscore(D) = 2$ by $\sigma(\mathbb{C}) = c_3$ and $\mathbb{C}$ changes

to $\{c_4\}$

Step 4: Because $\mathbb{C} = \{c_4\}$, the algorithm terminates.

Finally, based on the computed $cscore$ values, the C-AUM heuristic predicts, in the $cscore$ order, which apps will launch soon. Because a lower $cscore$ for an app means that the app is strongly related to the most recent apps launched, the C-AUM heuristic decides that the apps with lower $cscore$ values are likely to be launched in the near future.

The time complexity of building a C-AUM model for given $S = \langle a_1, a_2, \ldots, a_l \rangle$ can be computed as follows:

(i) Computing the cluster affinity values for all apps in $\mathbb{D}_S$: $O(l^2)$

(ii) Building clusters using the *single-linkage clustering* algorithm: $O(|\mathbb{D}_S|^2)$, because the time complexity of the *single-linkage clustering* algorithm is $O(|N|^2)$ for given N nodes.

(iii) Computing the $cscore$ for all apps in $\mathbb{D}_S$: $O(|C| \cdot |\mathbb{D}_S|)$

Since the app sequence length l is much larger than $|\mathbb{D}_S|$, the time complexity of building a C-AUM model can be approximated by $O(l^2)$. Although l can be continuously increasing, we have observed that a reasonable large constant l gives accurate $cscore$ values over when the exact l is used. When the constant app sequence length of 2500 is used, there are less than 5% differences on the $cscore$ values compared with the exact l-based implementation. When a constant l is used, we can further optimize the implementation of the C-AUM heuristic. For example, when 50 distinct apps are used (i.e., $|\mathbb{D}_S| = 50$), on average, 20 updates of launch radiuses are sufficient whenever a new app is launched. Therefore, in practice, the time

complexity of the C-AUM heuristic is reduced to $O(|\mathbb{D}_S|^2)$. In our current implementation on the Nexus S, it took on average 11.43 ms to run the C-AUM heuristic every single time a new app was launched based on the real usage log, for the given $S = \{a_1, \ldots, a_{2500}\}$ and $|\mathbb{D}_S| = 50$ using $|\mathbb{A}| = 3$ in computing the $cscore$ values of Algorithm 1.

Chapter 6

AUM-based Launching Experience Optimization Technique

6.1 Motivation

As a specific example of inter-app personalized optimization, in the remainder of this paper, we present an app launching experience optimization technique based on the app usage models. Considering that all user interactions at smartphones start by launching an app, a quick app launching without a noticeable delay is a prerequisite of a good experience. In this paper, we broadly define *app launching experience* as a type of user experience related to app launching in general.

As mentioned in Section 1.1.2, most smartphone systems such as the Android platform do not immediately terminate apps when a user no longer interacts with the apps in the foreground. Instead, for better launching experience, inactive apps are kept as cached apps in the main memory of smartphones. The cached apps are effectively terminated when the systems decide that they need more memory. Under this policy, app launching can be categorized in two types: a *hot start* and a *cold start*. If an app is restarted by simply restoring its previous state already kept in the memory, we call it the hot start. On the other hand, the cold start of an app happens when the

app is launched for the first time, or sometimes when the app is re-launched after an eviction from the memory. We use the terms *restart* and *cold start*, interchangeably.

When a cold start of an app occurs, it can adversely affect app launching experience over a hot start often with a user-perceived delay. Furthermore, the cold start is less energy-efficient and fails to return to the most recent execution state of the app. Considering the negative impact of the cold start on app launching experience, it is important to reduce the number of cold starts.

In order to better motivate our proposed app launching experience optimization technique, we present quantitative analysis of the impact of cold starts on performance, energy, and state preservation.

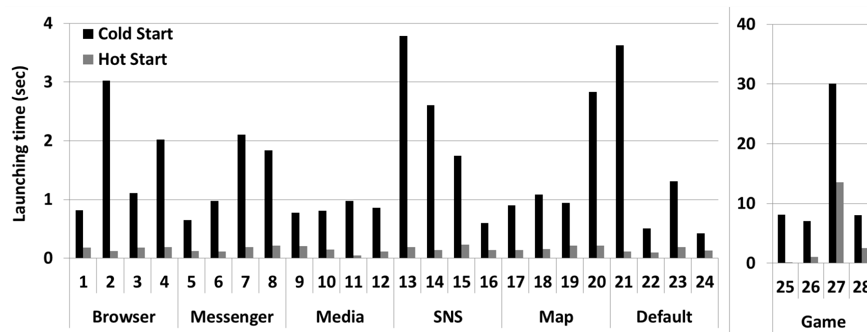


Figure 32: Launching time differences between hot and cold starts.

6.1.1 Impact of Cold Starts on App Launching Experience

Launching Time Difference between Hot and Cold Starts

In order to understand performance penalty associated with a cold start, we have measured the launching times of 28 Android Apps, which can be divided into 7 categories: Browser, Messenger, Media, SNS, Map, Game, and Default. The Default category denotes the apps which are supported by Google such as the calculator app, the market app, the default mail client, and the calendar app. While the launching start time can be accurately measured by monitoring when an intent to launch an app is received, it is difficult to precisely measure the launching completion time because many apps start to react users' input for better interactive user experience well before their launch procedure is completed. We thus define the launching completion time as the first moment the application becomes responsive. An in-house tool [23] was developed to measure launching times from this new definition.

Figure 32 compares the launching time of the apps for the hot and the cold start. The X-axis and the Y-axis denote various Android apps with their category and the launching time, respectively. Since the launching times of the apps in the Game category are significantly longer than others, they were presented using a different scale. As shown in Figure 32, the launching time of the cold start is on average 9 times longer than that of the hot start except for the apps in the Game category. For the apps in the Game category, the ratio between the launching time of the hot and cold starts is smaller than the other categories. However, the launching time difference between the hot and cold start is by up to 16.5 seconds, which is obviously too long for most users. (Note that a response delay of more that 1 second can make user uncomfortable [24].) The results show that it is important to reduce the number of cold starts in order to avoid a significant penalty in the launching time.

Energy Consumption Difference between Hot and Cold Starts

As mentioned above, a cold start incurs additional overheads, including process creation, file reads, network connections, accompanied by increased time delays. Since energy consumption depends on both the activities of each component in the device and the time spent, we can straightforwardly infer that the energy consumption in a cold start is much higher than a hot start. In order to understand exact differences on energy consumption, we measured the energy consumption of each app during its cold start and hot start using a power measurement environment similar to that of used in [25].

Figure 33 shows a snapshot of changes in measured currents during

the launching process of a hot start and a cold start of the same app. Since the supply voltage to the device was fixed in our measurement environment, Figure 33 shows differences in power consumption. In Figure 33, the X-axis and the Y-axis represent time in millisecond and the electrical current, respectively. As shown in Figure 33, the extra energy consumption caused by the cold start is observed significantly higher than that of the hot start.

State Loss in Cold Starts

For better app launching experience, it is important to resume from the previous state of an app when the app is launched again. Although most smartphone SDKs support ways to preserve the current state of an app when it is terminated, however, how to employ state preservation support in the app is entirely up to developers. In order to understand the impact of cold starts on the state preservation, we verified the degree of the state preservation for 60 apps. For quantitative analysis, we divided 60 apps into three categories: full state preservation, partial state preservation and no state preservation. When an app is restarted exactly at the same previous state, the app is classified into the full state preservation. If an app is launched with the same *Activity* as the app was terminated, the app belongs to the partial state preservation category. As shown in Figure 34, 62% of the apps analyzed support some degree of state preservation. Considering that hot starts always preserve the previous state, it is important to reduce the number of cold starts so that users can return to the same previous state.

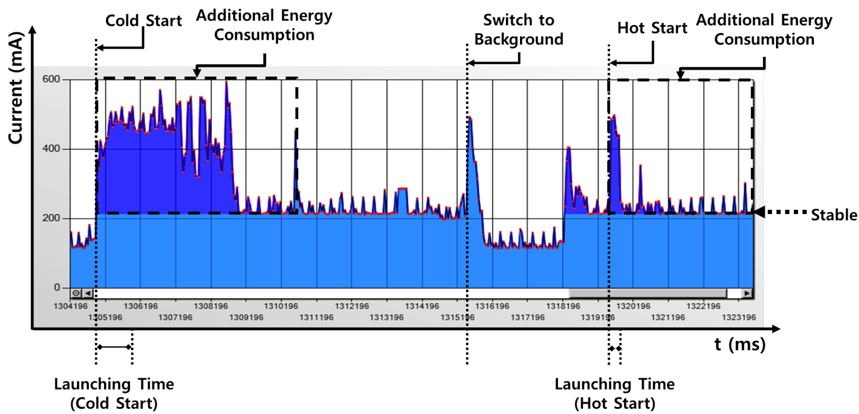


Figure 33: Current changes during the launching process.

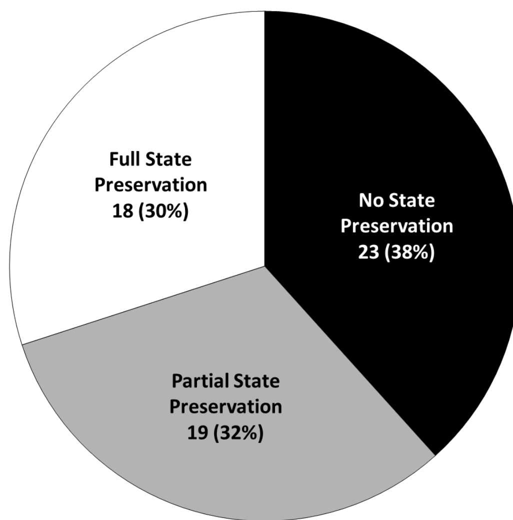


Figure 34: A breakdown of 60 apps based on the degree of state preservation.

6.1.2 Android Task Management Scheme

In an earlier version of Android (before 2.2 Froyo), because the device memory was limited, apps have to be terminated when available memory

is not sufficient. However, as the size of the device memory continues to increase, app termination occurs less frequently. As a result, a large number of apps and their processes were resident in the memory, which can be often a burden to memory management. In order to avoid this burden, a new task killing policy was added to the Activity Manager in the Android 2.2 platform. It limits the number of the background processes (which are called as hidden apps[1]) lower than a predefined maximum number whose default value is set to 15.

When the number of the background processes becomes larger than the predefined maximum number, the task killing policy proactively kills the excess number of hidden apps. In the Android framework code, this predefined maximum number is named as MAX_HIDDEN_APPS. From our analysis of the collected app usage logs from 21 users, we have found that this policy is the main source of terminating background apps.

6.1.3 Problem of the LRU-based Task Killer

App Restart Ratio

In order to evaluate the default LRU-based task killing policy, we need to know that how often each app is restarted. To this end, we introduce a metric named "restart ratio", which is defined as a fraction of the total number of app relaunches over the total number of app launches. The restart ratio is used to evaluate the effectiveness of a task killing policy. If the restart

[1]Note that not every process in the background state is classified as a hidden app, because there are several special background processes which always have to reside in memory.

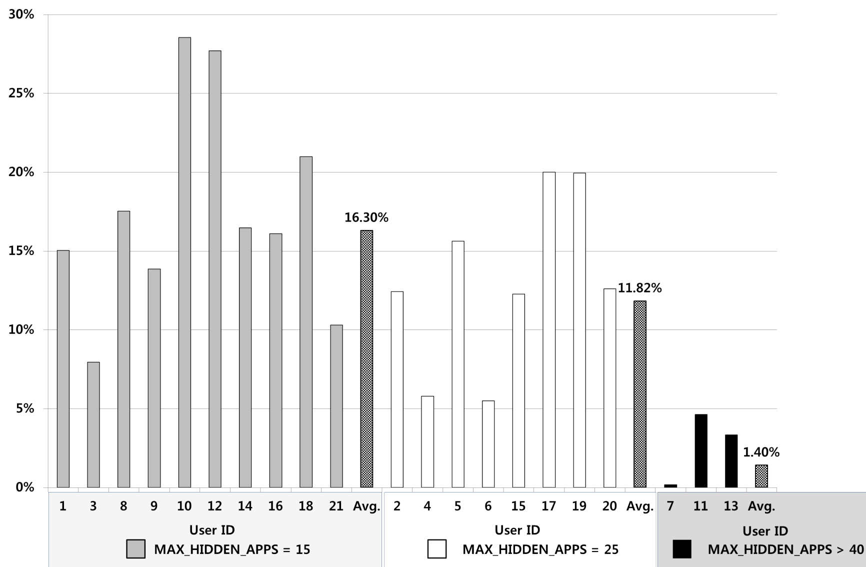


Figure 35: Restart ratio distributions of 21 users.

ratio is high, the user will suffer from a poor user experience caused by frequent app relaunches.

Figure 35 shows a distribution of the app restart ratios for 21 users under the LRU-based policy. When MAX_HIDDEN_APPS is set to the default value, 15, the average restart ratio is 16.30% and only one user (*users 3*) has experienced the restart ratio less than 10%. Figure 35 also shows the effect of the MAX_HIDDEN_APPS value on the restart ratio. In case of a very large MAX_HIDDEN_APPS value (i.e., > 40), the restart ratio improves very quickly as shown for *users 7, 11, and 13*. However, as explained earlier in Section 6.1.2, a large number of background processes will incur other overhead in memory management with a risk of continuous memory leaks from poorly-behaving tasks. Therefore, it is important to minimize the restart ratio under a small MAX_HIDDEN_APPS value.

User Context-oblivious Task Termination Problem

The Android's task killing policy selects victims under the assumption that the hidden apps placed near the LRU location are less likely to be started again. That is, the killing policy is based on the recency of app usages. However, suppose that there are specific apps mostly used in a certain user context, and this user context repeatedly appeared in the past app usage sequence. In this case, since the LRU-based policy only considers the recency of app usage, it cannot quickly adapt to changing user contexts. For example, if a user changes from *Context A* to *Context B*, apps used in *Context B* may be selected as termination candidates in the LRU-based policy because they were not used recently even if they are likely to be launched in a near future. For this reason, the performance of the LRU-based task killing policy deteriorates quickly. In addition, each app restart leads to a user-perceived delay, extra energy consumption, and state loss as explained in Section 6.1.1. Therefore, in order to avoid such extra restarts, it is necessary for a task killer to recognize the app's usage pattern prior to making a decision.

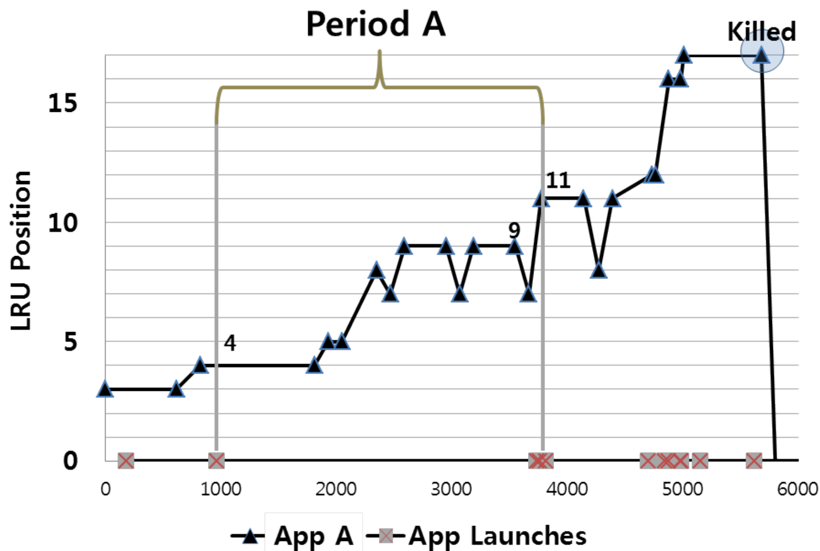


Figure 36: Changes in LRU stack positions of *App A* over time.

LRU Stack Pollution Problem

One of the main sources for a large number of app restarts in Android comes from the services and app widgets. Figure 36 shows how the LRU stack position of *App A* varies until *App A* is killed by the LRU-based task killer. Although none of other apps were launched during the `Period A`, the LRU stack position of *App A* switches from the position 4 to the position 11 where the position 0 is the MRU position. This demotion in *App A*'s LRU stack position is from the executions of app widgets and services. In our log analysis, *App A* was a very *unlikely* candidate to be killed by the LRU-based task killer because it was launched very frequently over the entire log collection time. Therefore, the LRU stack position demotion in the `Period A` was the main reason of *App A* being killed.

6.2 AUM-based Launching Experience Optimization

6.2.1 App Usage (AU)-aware Task Killer

In order to avoid the app relaunch problems of the LRU-based task killer policy, we have developed the app usage (AU)-aware task killer using two usage models, P-AUM and C-AUM. As mentioned above, the Android platform employs the LRU-based task killing policy to limit the number of the background processes. The LRU-based task killing policy is triggered when the number of the background processes exceeds the predefined maximum number. The LRU-based policy selects victims under the assumption that the processes placed near to the LRU location are less likely to be reused. In order to solve the problems with the LRU-based policy (discussed above), we implemented the AU-aware task killer, which selects a victim based on our AUM.

Algorithm 3 AUM-based victim task selection and termination

```
for each app x in BackgroundAppList do
    score ← AUM.getScore(x)
    x.setScore(score)
    ScoredAppList.add(x)
end for
AppListSortedByScore ← stableSortByScore(ScoredAppList)
while (AppListSortedByScore.size() > MAX_HIDDEN_APPS) do
    victimApp ← getHighestScoreApp(AppListSortedByScore)
    killProcess(victimApp)
end while
```

Algorithm 3 describes how the AU-aware task killer terminates apps. Based on the score of each app, which was assigned by our AUM, a new

app list, AppListSortedByScore, is constructed in a non-decreasing order. Since several apps can have the same score, a stable sort algorithm is used to maintain the LRU order in case of ties. When the number of the maintained background apps exceeds MAX_HIDDEN_APPS, the app with the highest score is selected as a victim. This selection and termination process is repeated until the number of the background apps drops below the MAX_HIDDEN_APPS value.

In case of applying the P-AUM heuristic, considering both the computational complexity and the prediction accuracy, we use the most recent 4 apps to find similar patterns in the past app sequence. (According to our experimental results, selecting the most recent 3 apps was not sufficient to find similar patterns correctly because 3 apps cannot adequately represent the current execution context.)

6.2.2 App Usage (AU)-aware Prelauncher

As another approach to improve app launching experience, we developed a prelaunch mechanism based on our AUM. To this end, we implement the AU-aware prelauncher, which launches apps in advance of the actual launches by a user. In order to avoid any interference with active apps, prelaunching is only considered when there is a long screen-off idle time.

In our implementation, we manage an additional pool, for prelaunched apps apart from the existing process list, as shown in Figure 37. The number of currently prelaunched apps, called as $PoolCount$, is decided depending on the amount of available memory, as in Equation 6.1.

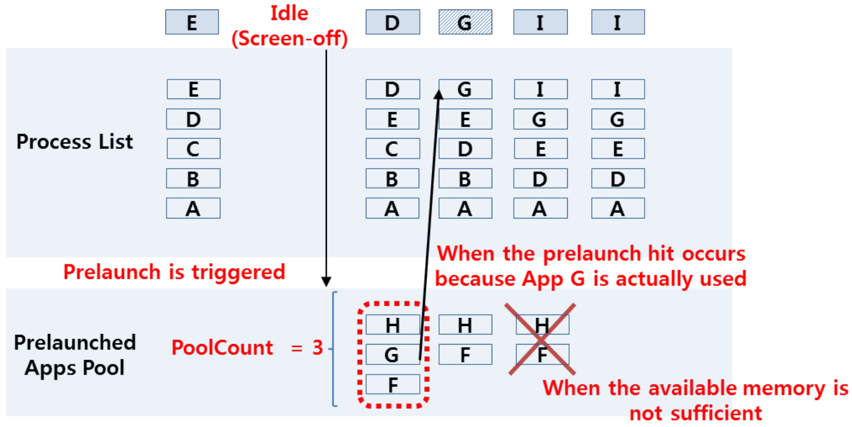


Figure 37: The overview of the AU-aware prelauncher.

$$PoolCount = \frac{AvailMem - SpareMem}{AvgMemOfApp} \tag{6.1}$$

$AvailMem$ is the amount of available memory in the system and $SpareMem$ denotes the amount of specially provisioned memory in order to prepare for a possible fluctuation in memory usage, respectively. $AvgMemOfApp$ is the average memory size which each app occupies when it is running. After $PoolCount$ is decided, prelaunched apps are placed in the pool for the prelaunched apps. For example, in Figure 37, H, G, F are prelaunched apps. When available memory is not sufficient, all prelaunched apps in the pool are terminated. Therefore, when a certain prelaunched app is actually launched by the user, the app should be moved to the process list in order to prevent the prelaunched apps from an unintended termination. In the example, G is moved to the process list (because the user has actually launched G.) On the other hand, both H and F are terminated to reclaim

memory needed.

6.3 Experimental Results

6.3.1 Experiment Environment

In order to evaluate the efficiency of the proposed framework and optimization techniques, we implemented the POA framework and AUM-based launching experience optimization techniques in the Android platform version 2.3 (Gingerbread), running on the Nexus S Android reference smartphone. In addition to the proposed P-AUM and C-AUM heuristics, we also implemented two more heuristics, LFU and Oracle, to the reference smartphone. (The task killing mechanisms based on LFU and Oracle will be explained in the next subsection.) The prelaunching technique, which was described in Section 6.2.2, was also implemented in the real smartphone platform. In our experiments, we have used the log replayer tool for quickly executing the app sequences extracted from the user logs.

The MAX_HIDDEN_APPS value can be varied according to the hardware specifications of a smartphone. In the case of the usage logs whose MAX_HIDDEN_APPS value is more than 15, it is difficult to reproduce the similar realistic execution environments to the app usage logs collected from the active smartphone users. This is because we chose the Nexus S smartphone, whose hardware specifications are different from the smartphones which had adopted the MAX_HIDDEN_APPS value more than 15, for the evaluation. Out of 21 user logs we have collected, we have selected four usage logs as they represent typical usage scenarios in terms of the restart

Table 5: Summary of four representative user logs.

	Used Apps	Total App Launches	Restart Counts	Restart Ratio (Logged)	Restart Ratio (Replayed)	MAX_HIDDEN_APPS
User 1	52	2454	515	21.0%	16.9%	15
User 2	42	956	273	28.6%	23.3%	15
User 3	58	1329	233	17.5%	12.5%	15
User 4	35	1341	221	16.5%	10.9%	15

ratio, the usage pattern, and the MAX_HIDDEN_APPS value. In detail, the usage logs of which the restart ratio is the maximum and the minimum ratio were omitted. We also excluded the usage logs of which a large number of the restarts came from one-time use apps or newly launched apps. Table 5 summarizes the main characteristics of four usage logs.

In order to reproduce realistic execution environment as real executions, we executed both the app widgets and the services between app launches in a controlled fashion. We determined when and how many app widgets and services will be launched based on the analysis of the LRU stacks collected from the logs.

6.3.2 Results of Task Killing Mechanism Optimization

Restart Ratio Comparisons

Figure 38 compares the restart ratios of five different task killing policies for four representative users. As shown in Figure 38, our task killing mechanism optimization based on the P-AUM and C-AUM heuristics can reduce the restart ratio by up to 74.4% and 78.4% compared to the LRU policy, respectively. In addition, the optimization based on the C-AUM heuristic

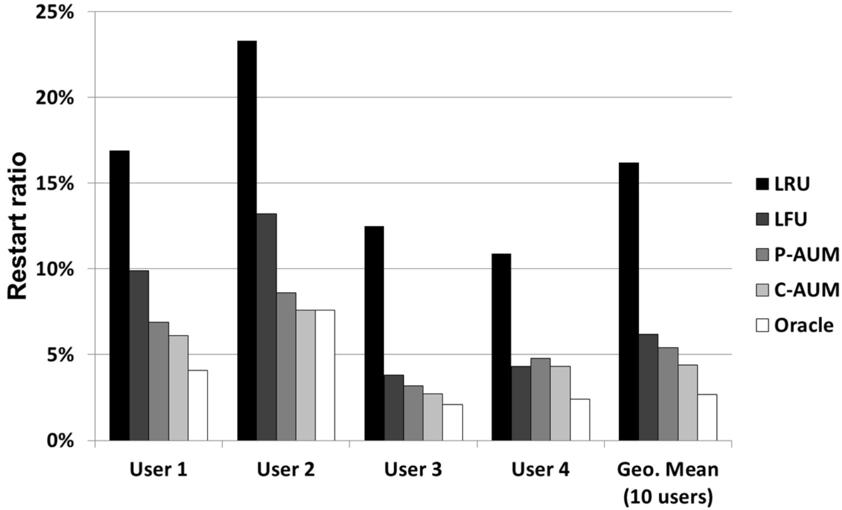


Figure 38: Restart ratio comparisons of five policies.

always outperforms that based on the P-AUM heuristic. In fact, C-AUM performs close to Oracle (which, at the time of a victim task selection, assumes a complete future knowledge on future app launches). LFU, which selects a victim task based on the frequency of app launches, also outperforms LRU.

In order to give a more intuition behind why the C-AUM based policy works better over LRU and LFU, we show a detailed trace of one app (say, *App A*) as an example of microscopic analysis. We define the rank of an app as its position as managed by each policy. The lower the rank of an app is the more important the app is. As shown in Figure 39, the LRU policy is vulnerable to the LRU stack pollution problem (as discussed in Section 6.1.3). In the pollution period, though no apps are launched by a user, the rank of *App A* is continuously decreased. *App A* is terminated by the LRU policy just before being re-used. In the LFU-based policy, since *App A* was not fre-

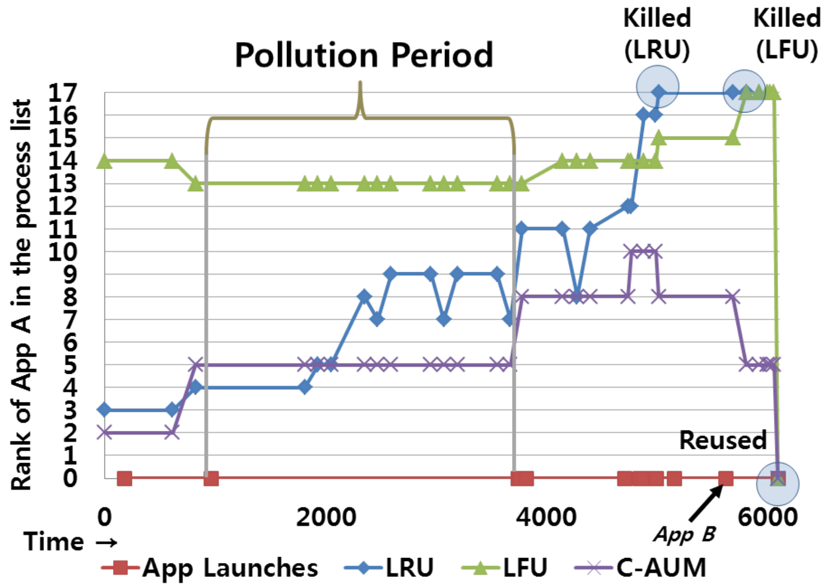


Figure 39: Changes in the rank of *App A* over time.

quently used, the rank of *App A* was high, so it was terminated when another app (i.e., *App B*) was launched. On the other hand, the C-AUM based policy can maintain *App A* as an important app by observing the current sequence of app launches. Moreover, the rank of *App A* is increased just before it is reused because *App B*, which was strongly related to *App A* was launched.

Although the restart ratio is a useful measure for comparing different policies, it alone does not tell the complete picture. For example, for the same number of app restarts, user experience may be completely different. When the same app is relaunched frequently in a short period of time, the user will feel very uncomfortable. Therefore, we defined another metric, *weighted restart count*, which gives a higher cost if an app is restarted in the same interactive session. (An interactive session is defined to be an interval

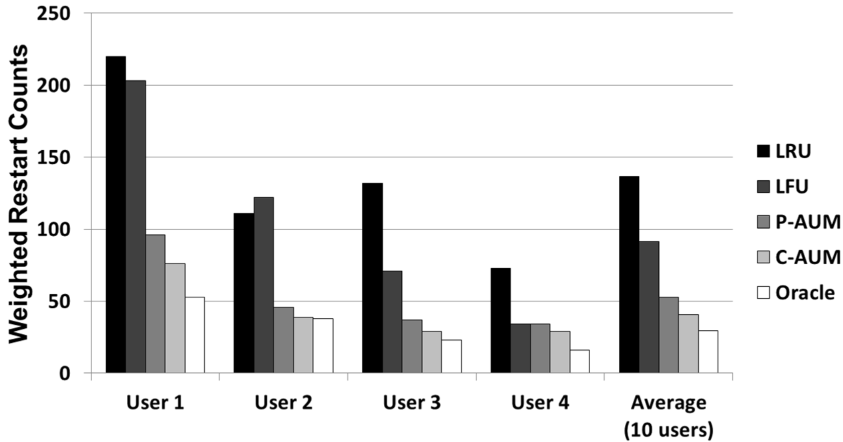


Figure 40: Weighted restart count comparisons of five policies.

between two consecutive screen-off points.) Figure 40 compares *weighted restart counts* for five policies. Compared to the restart ratio comparison result of Figure 38, the LFU policy performs noticeably poorer than our P-AUM and C-AUM. This poor weighted restart count of the LFU policy indicates that LFU cannot adequately handle particular app usage patterns as P-AUM and C-AUM do. Under LFU, it is a lot more likely that a user suffers a long launching time when the user is actively involved in an interactive session.

Impact on App Launching Experience

As mentioned in Section 6.1.1, when a cold start of an app occurs, it can adversely affect app launching experience over a hot start from aspect of a user-perceived delay, extra energy consumption, and state loss. In order to verify the impact of cold starts on the launching experience, we evaluated the launching time, additional energy consumption, and state loss ratio of

each task killing policy. Figure 41 shows how our AUM-based techniques influence the launching time for four representative users. As anticipated, the C-AUM based optimization technique can reduce the launching time by up to 40% compared to the default Android policy. The results show that it is possible to improve the launching experience by increasing the number of hot starts so that the users can start their apps without any delays because the launching time of the hot start is negligible.

Figure 42 presents the impact of each task killing policy on the energy consumption when the devices are assumed to connect the network via WiFi connections. The results show that the proposed optimization techniques can achieve on average an improvement in the energy consumption of 19.79% and 22.48%, respectively.

Figure 43 compares state loss ratios of four policies. The state loss ratio is the fraction of all app launches that lead to any state loss. In the LRU-based policy, the previous state is not preserved once in ten launches. On the other hand, the state loss occurs on average 3.17% and 2.52% of app launches in our proposed techniques, respectively.

Evaluation of Termination Decision Accuracy

As mentioned in Section 6.1.2, when the number of the hidden apps exceeds MAX_HIDDEN_APPS, the Android platform selects a victim among the hidden apps, shown as candidates in Figure 44, which are currently in the process list. Since a task killing policy operates under the assumption that the selected victim will not be re-used for a long time, counting the number of apps launched before the victim app is re-launched in the future can give

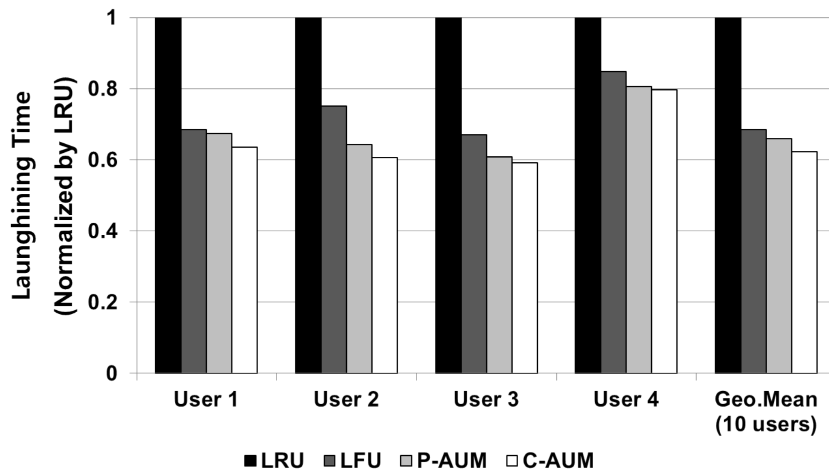


Figure 41: Normalized launching time comparisons of four policies.

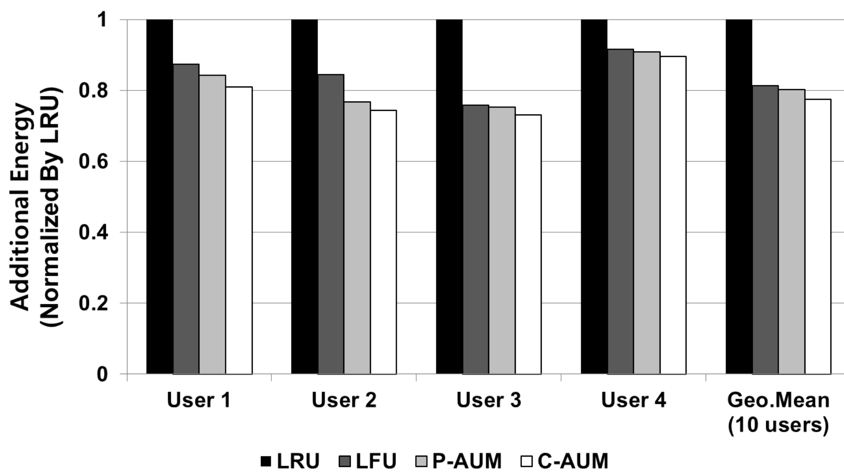


Figure 42: Normalized energy consumption comparisons of four policies.

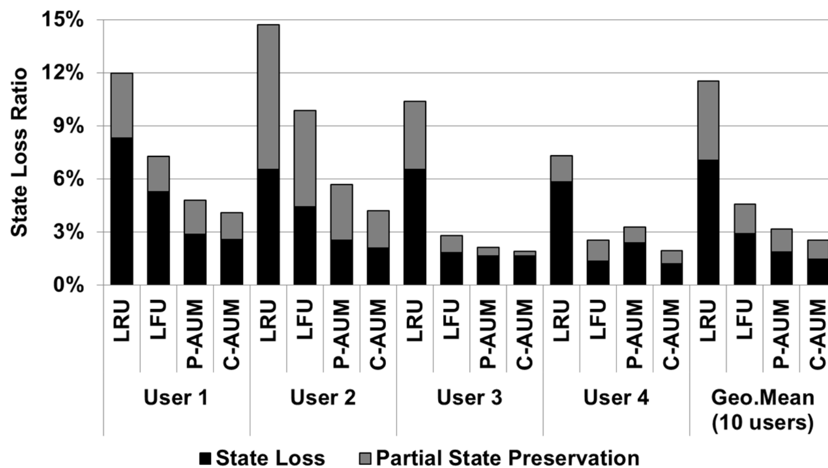


Figure 43: State loss ratio comparisons of four policies.

us an efficient metric to evaluate the accuracy of a task killing policy. We thus defined an evaluation metric, termination accuracy grade, as the number of distinctive candidate apps which appear during the period between the termination point and the restart point of the victim app in the app usage log sequence. Since the decision can affect only the restart of candidates, the number of the distinctive candidate apps launched during this period is considered as the termination accuracy grade.

Figure 44 illustrates how to compute the termination accuracy grades. When *App F* is launched, *Apps A*, *B*, *C*, *D*, and *E* are the candidates for a victim. In this example, the termination accuracy grade of *App A* is 4 because the candidates will be re-launched in the order of *App C*, *B*, *E*, *A*, and *D*.

In order to evaluate the accuracy of termination decisions of different policies, termination accuracy grades are calculated every termination deci-

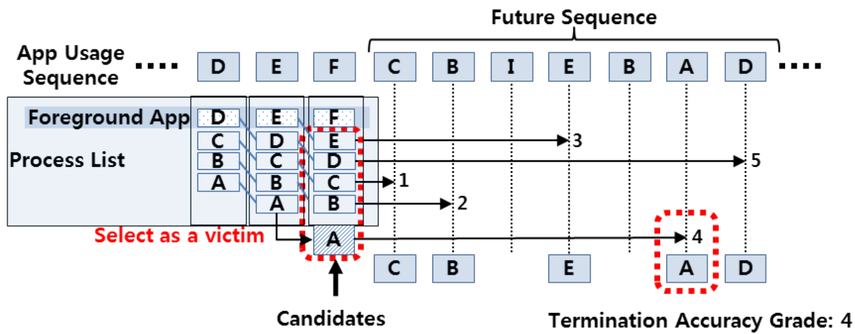


Figure 44: An example of computing termination accuracy grades.

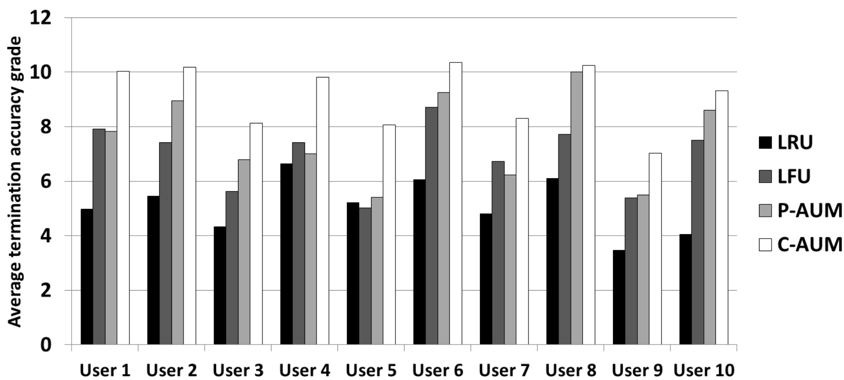


Figure 45: Comparisons of the average termination accuracy grade of four policies.

sion times for the different task killing policies, including the LRU-based, LFU-based, P-AUM based, and C-AUM based policies. As shown in Figure 45, the average termination accuracy grade of the C-AUM based policy is always larger than the other policies, thus the C-AUM based policy makes more intelligent decisions on future app usage.

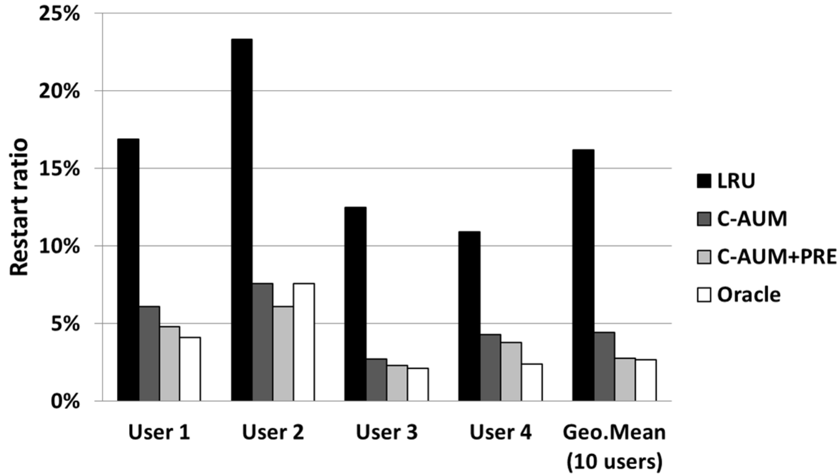


Figure 46: The effect of the prelaunching technique on the restart ratio.

6.3.3 Results of Prelaunching Technique

Figure 46 shows the effect of prelaunching on the restart ratio. The C-AUM based policy combined with the proposed prelaunching technique, shown as C-AUM+PRE in Figure 46, reduces the restart ratio by up to 21.3% over the C-AUM based policy.

It is interesting to note that the C-AUM+PRE policy even outperforms the oracle policy (the Oracle policy in Figure 46) that makes task termination with the complete future knowledge on app launches. This is because the C-AUM+PRE policy can launch more than MAX_HIDDEN_APPS apps at the same time when there are enough memory available to do so. On the other hand, the Oracle policy is limited to MAX_HIDDEN_APPS apps.

Chapter 7

Conclusions

7.1 Summary and Conclusions

As truly personalized and interaction-oriented devices, understanding and analyzing distinctive inherent characteristics of smartphones provide a new novel opportunity for optimizing various system design requirements. Therefore, user-specific high-level information, when properly analyzed and managed, can provide valuable hints for smartphone system optimization.

In this dissertation, by taking advantage of high-level information from the smartphone users, we proposed several user-centric optimization techniques to satisfy various design requirements of smartphones such as energy efficiency, effective thermal management and rapid responsiveness without any performance degradation.

First, we have presented **ura**, a user-perceived response time analyzer for Android-based smartphones, and a new CPU power management framework based on **ura**. By taking advantage of the on-line identification of the user-perceived response time from **ura**, our proposed CPU power management framework allows more aggressive low-power techniques to be applied to smartphones. In order to demonstrate the effectiveness of our proposed framework, we have developed the `oninterval` *cpufreq* governor. Based on understanding and analyzing the user-perceived response time, our pro-

posed `oninterval` governor could make aggressive DVS decisions without any negative effect on user experience. Our experimental results show that the `oninterval` governor can save the CPU power consumption by up to 65.6% over the Linux default `ondemand` governor.

Second, we have proposed SmartDTM, a new dynamic thermal management technique for smartphones. Based on the **ura**'s on-line identification, out proposed SmartDTM ensures the quality of user experience during the execution of display-sensitive parts without any thermal violations which are predefined by the system. Our experimental results show that the proposed technique can improve the user-perceived performance by up to 37.96% over the Android's default DTM policy.

Lastly, we have presented POA, a personalized optimization framework for Android smartphones. Taking advantage of the fact that smartphones are truly personal devices, POA builds user's app usage models during run time, and enables more advanced and effective optimizations for smartphones. Moreover, we have developed a couple of app usage models which can be used in predicting typical smartphone user's future app usage tendencies. Based on the app usage models, we have developed an app launching experience optimization technique which effectively reduces expensive app restarts so that a user can launch apps with smaller user-perceived delays, while reducing energy consumption with better state preservation. Experimental results showed that our optimization technique implemented on Android smartphones reduced the number of unnecessary app restarts by up to 78.4% over the Android's default policy.

7.2 Future Work

Although our results show dramatic benefit of user-centric optimization techniques, there are several possible improvements and opportunities that we have not fully explored.

7.2.1 Improving Prediction Accuracy of the AUMs Using Context Information

In this dissertation, we have developed two AUMs that correctly capture typical smartphone user's application usage patterns. In order to build the AUMs on the relatively poor computing resources of the smartphone, our proposed AUMs exploit comparative simple methods such as Damerau-Levenshtein distance and single-linkage clustering algorithm to identify application usage patterns. Since only proper prediction accuracy for the future app usage is required in our AUM-based optimization techniques, exploiting those simple methods is enough to meet the design requirement.

However, in order to apply the proposed AUMs to the other inter-app optimization points, demanding more detailed information about the future app usage, the prediction accuracy of the proposed AUMs should be improved. For example, when a particular app needs massive size of contiguous memory allocation, many cached apps should be terminated at once even if the number of cached apps does not exceed the predefined maximum number. As a result, there are limited number of cached apps in the system memory after the app which requires the contiguous memory allocation is launched. In that case, if we can proactively fill the system memory

with the apps which are more likely to use in the near future, it is possible to provide a higher launching experience. In order to maximize the effectiveness of the prelaunching, it is required to select a much larger number of apps as the candidates for the prelaunching compared with that of our AUM-based prelaunching technique. Therefore, the prediction accuracy is critical in the performance of the prelaunching policy. Considering that our proposed AUMs only focus on exploiting the past app usage log, we can extend the proposed AUMs to include different types of context information (e.g., location and time) for the improvement of the prediction accuracy. For example, in FALCON, in addition to the relations between the apps in the past app usage log, locations and times strongly correlated with distinctive app usage patterns are also exploited to build the AUM. However, since categorizing all the collected locations into several semantic places is relatively heavy to perform on the smartphone, simplified heuristics are required to exploit the location information without reliance on the external servers or cloud services. One possible way of exploiting the location information without the reliance on the external server is to use the accelerometer sensor. By combining the accelerometer and GPS data, we can easily categorize the collected locations into two state, staying and moving. Since all the locations which belong to the moving state can be excluded from the collected locations, it is possible to significantly reduce the number of locations to be analyzed so that the AUM can exploit the location context with lower computation overhead.

7.2.2 Integrated Intra- and Inter-App Approaches for User-Centric Optimizations

The proposed optimization techniques in this dissertation take advantage of user-centric high-level information. The proposed user-centric optimization techniques can be categorized into two classes, intra-app and inter-app optimization techniques. The intra-app optimization techniques mainly focus on optimizing the system requirements within an individual app boundary while the inter-app optimization techniques make more intelligence operational decisions on an app-by-app basis. In this dissertation, the former ones are **ura** and **ura** optimization techniques and the latter ones are POA and AUM-based optimization techniques.

If the proposed intra- and inter-app optimization techniques is integrated, the quality of user experience can be further improved. For example, our current AUM-based app launching optimization techniques are designed under the assumption that all the costs of app restarts are same. However, the costs of the app restarts, particularly in terms of user-perceived delays and energy consumption, are quite different from each other in practice. Therefore, by taking advantage of the on-line identification of the user-perceived response time from **ura** as a weight of each app restart, the proposed AUM-based app launching optimization techniques can be extended to distinguish apps with long user-perceived restart times from ones with short user-perceived restart times. For apps with very short restart times, it may be better to terminate them instead of keeping them in memory as cached apps. Furthermore, our AU-aware prelauncher can be also extended

to select the candidates for the prelaunching by exploiting the user-perceived restart time information.

7.2.3 User-Centric Optimizations for The Other Design Requirements

In this dissertation, our AUM-based launching experience optimization techniques have only focused on the termination cases when the number of background apps exceeds the predefined maximum number. However, we can apply our proposed AUMs to the other design requirements. For example, in Android, there are many conditions that can result in termination of the cached apps. Therefore, in order to completely improve the app launching experience, it is highly required to make the other termination decisions of the cached apps also under the proposed AUMs. For example, the low memory killer is a privileged process, which terminates the cached apps when the available memory is lower than the predefined size. Although the termination decision of the low memory killer can significantly affect the quality of user experience, only LRU-based policy is currently used. If we can exploit the proposed AUMs, the low memory killer can make more user-centric decisions, resulting in user experience improvement.

In addition, the **ura**-based power optimization framework can be also further extended to the manage the energy consumption of different system components by exploiting the user-perceived response time. For example, the operating frequencies of memory-interface, bus and GPU in the smartphone are mostly decided by the recent utilization of those devices. As a

result, existing mobile OS power management schemes (e.g., *devfreq* governors of the Linux kernel) cannot quickly adapt to changing the impact of the current execution on the quality of user experience. Based on understanding and analyzing the user-perceived response time, our proposed power optimization framework also employs more aggressive low-power techniques in the user-oblivious response time interval for those different system components.

Bibliography

[1] D. Garlan, D. P. Siewiorek, and P. Steenkiste, “Project Aura: Toward Distraction-Free Pervasive Computing,” *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 22-31, 2002.

[2] M. Satyanarayanan, “Pervasive Computing: Vision and Challenges,” *IEEE Personal Communications*, vol. 8, no. 4, pp. 10-17, 2001.

[3] Wireless Intelligence, “Smartphone Users Spending More ‘Face Time’ on Apps than Voice Calls or Web Browsing,” https://www.wirelessintelligence.com/analysis/2011/03/smartphone-users-spending-more-face-time-on-apps-than-voice-calls-or-web-browsing, 2011.

[4] Digitizor Media&Web, “Android Stats: 200K Market Apps, 400K New Activations Daily, Malware Up By 400%,” http://digitizor.com/2011/05/11/android-stats, 2011.

[5] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, “Diversity in Smartphone Usage,” in *Proceedings of the International Conference on Mobile Systems, Applications, and Services* (MobiSys), 2010.

[6] X. Bao, M. Gowda, R. Mahajan, and R. R. Choudhury, “The Case for Psychological Computing,” in *Proceedings of the International Work-*

shop on Mobile Computing Systems and Applications (HotMobile), 2013.

[7] A. Shye, B. Scholbrock, G. Memik, and P. A. Dinda, "Characterizing and Modeling User Activity on Smartphones: Summary," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS), 2010.

[8] T. M. T. Do, J. Blom, and D. Gatica-Perez, "Smartphone Usage in the Wild: A Large-Scale Analysis of Applications and Context," in *Proceedings of the International Conference on Multimodal Interaction* (ICMI), 2011.

[9] N. Kiukkonen, J. Blom, O. Dousse, D. Gatica-Perez, and J. Laurila, "Towards Rich Mobile Phone Datasets: Lausanne Data Collection Campaign," in *Proceedings of the ACM International Conference on Pervasive Services* (ICPS), 2010.

[10] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in Smartphone Usage," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services* (MobiSys), 2010.

[11] Microsoft, "Inside the Windows Vista Kernel," http://www.microsoft.com/technet/technetmag/issues/2007/03/VistaKernel, 2013.

[12] B. Esfahbod, "Preload - An Adaptive Prefetching Daemon," M.S. Dissertation, University of Toronto, 2006.

[13] J. Ryu, Y. Joo, S. Park, H. Shin, and K. G. Shin, “Exploiting SSD Parallelism to Accelerate Application Launch on SSDs,” *Electronics Letters*, vol. 47, no. 5, pp. 313-315, 2011.

[14] J. A. Baiocchi, and B. R. Childers, “Demand Code Paging for NAND Flash in MMU-Less Embedded Systems,” in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2011.

[15] Y. Joo, J. Ryu, S. Park, and K. G. Shin, “FAST: Quick Application Launch on Solid-State Drives,” in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2011.

[16] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, “Fast App Launching for Mobile Devices Using Predictive User Context,” in *Proceedings of the International Conference on Mobile Systems, Applications, and Services* (MobiSys), 2012.

[17] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, “LiveLab: Measuring Wireless Networks and Smartphone Users in the Field,” in *Proceedings of the ACM SIGMETRICS Performance Evaluation Review* (PER), 2011.

[18] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. 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), 2013.

[19] J. G. Cleary, and I. Witten, "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Transactions on Communications*, vol. 32, no. 4, pp. 396-402, 1984.

[20] X. Li, G. Yan, Y. Han, and X. Li, "SmartCap: User Experience-Oriented Power Adaptation for Smartphone's Application Processor," in *Proceedings of the Conference on Design, Automation and Test in Europe* (DATE), 2013.

[21] B. K. Donohoo, C. Ohlsen, and S. Pasricha, "AURA: An Application and User Interaction Aware Middleware Framework for Energy Optimization in Mobile Devices," in *Proceedings of the IEEE International Conference on Computer Design* (ICCD), 2011.

[22] M. Martins, and R. Fonseca "Application Modes: A Narrow Interface for End-User Power Management in Mobile Devices," in *Proceedings of the International Workshop on Mobile Computing Systems and Applications* (MobiSys), 2013.

[23] J. Lim, H. Kim, W. Song, and J. Kim, "LTmeter: An App Launching Time Analyzer for Personal Smart Devices," in *Proceedings of the International Conference on Ubiquitous Information Technologies & Applications* (CUTE), 2011.

[24] N. Tolia, D. G. Andersen, and M. Satyanarayanan, "Quantifying Interactive User Experience on Thin Clients," *IEEE Computer*, vol. 39, no. 3, pp. 46-52, 2006.

[25] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (CODE+ISSS), 2010.

[26] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh, "Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption," in *Proceedings of the ACM International Conference on World Wide Web* (WWW), 2012.

[27] V. Pallipadi, and A. Starikovskiy, "The Ondemand Governor," in *Proceedings of the Linux Symposium*, 2004.

[28] Android Open Source Project, "MonkeyRunner," http://developer.android.com/tools/help/monkeyrunner_concepts.html, 2013.

[29] X. Chen, Y. Chen, Z. Ma, and F. C. A. Fernandes, "How is Energy Consumed in Smartphone Display Applications?," in *Proceedings of the International Workshop on Mobile Computing Systems and Applications* (HotMobile), 2012.

[30] T. Nguyen, M. Mochizuki, K. Mashiko, Y. Saito, and I. Sauciuc, "Use of Heat Pipe/Heat Sink for Thermal Management of High Performance," in *Proceedings of the Annual IEEE Semiconductor Thermal Measurement and Management Symposium* (SEMI-THERM), 2000.

[31] J. M. Kim, Y. G. Kim, and S. W. Chung, ”Stabilizing CPU Frequency and Voltage for Temperature-Aware DVFS in Mobile Devices,” IEEE Transactions on Computers, vol. 64, no. 1, pp. 286-292, 2015.

[32] D. Brooks and M. Martonosi, ”Dynamic Thermal Management for High-Performance Microprocessors,” in *Proceedings of the International Symposium on High Performance Computer Architecture* (HPCA), 2001.

[33] Hardkernel, “ODROID-XU+E,” http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137463363079, 2013.

[34] V. I. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions and Reversals,” *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707-710, 1966.

[35] P. H. A. Sneat, “The Aplication of Computers to Taxonomy,” *Journal of General Microbiology*, vol. 17, no. 1, pp. 201-226, 1957.

초 록

최근의 스마트폰은 단순히 이동성이 있는 보조적인 수단으로서의 컴퓨팅 장치가 아닌 기존의 환경을 대체하기 위한 “손안의 데스크톱”의 위상으로서 빠르게 자리매김하고 있다. 스마트폰 환경에서는 사용자 경험이 매우 중시되기 때문에 사용자가 인지할 수 있는 성능 저하 없이 응용 프로그램을 구동하는 것은 물론 이를 동적이고 유려한 사용자 인터페이스와 함께 제공하는 것이 지극히 당연한 사용자 요구 사항으로 받아들여지고 있다. 따라서, 기존의 데스크톱 또는 랩톱 시스템에서처럼 스마트폰에도 높은 클록 속도를 갖는 고성능의 멀티코어 프로세서를 탑재하는 것이 일반적인 추세이며, 스마트폰에 탑재된 시스템 소프트웨어 또한 기존의 환경에서 널리 사용되고 있는 다양한 최적화 기법들을 적용함으로써 사용자가 인지할 수 있는 성능의 지연과 소비 전력 등을 최소화하고자 하는 실정이다.

한편, 스마트폰은 다수의 사용자가 아닌 한 명의 주도적인 사용자가 장치 및 응용 프로그램과 밀접하게 관계를 맺고 있으며, 계속되는 상호작용을 통하여 실행 흐름을 결정하는 등 기존의 컴퓨팅 환경과 그 사용 환경이 뚜렷하게 구분된다. 하지만, 현재 스마트폰의 소비 전력, 발열, 반응성 등을 최적화하기 위해 적용된 다양한 하드웨어 및 소프트웨어 수준의 최적화 기법들은 앞서 언급한 스마트폰만의 독특한 특성을 전혀 고려하고 있지 않으며, 기존 시스템의 기법들과 그 동작 원리나 방식에 있어 큰 차이가 없는 것이 대부분이기 때문에 사용자에게 최적의 사용 경험을 제공하지 못하는 문제를 안고 있다.

본 논문에서는 스마트폰에서의 소비 전력, 발열, 반응성과 같은 다양한 설계 요구 조건을 달성하기 위한 새로운 접근 방법으로서 사용자 중심의 최적화 기법을 제안한다. 제안한 기법들은 스마트폰 사용자에 대한 깊이 있는 분석을 통하여 끌어낸 높은 수준의 정보를 다양한 설계 요구 조건의 최적화에 적용하는 것이 기본 동작 원리이며, 이를 바탕으로 사용자 경험을 극대화하는 것을 목표로 한다.

첫 번째로, 스마트폰 환경의 응용 프로그램을 대상으로 사용자 중심의 새로

운 성능 평가 지표인 사용자 인지 반응 완료 시간을 제안한다. 사용자 인지 반응 완료 시간은 사용자가 명시적으로 입력을 인가한 시점부터 이러한 입력에 대한 반응으로써 UI가 갱신 완료되는 시점까지의 소요 시간이다. 사용자 인지 반응 완료 시점을 기준으로 그 앞의 실행 구간은 시스템의 성능이 사용자 경험에 큰 영향을 주지만, 그 뒤의 실행 구간은 시스템의 성능과 사용자의 경험 사이에 큰 연관 관계가 없다는 특성을 갖는다. 이러한 특성을 활용하여 시스템의 성능이 사용자 경험에 주는 영향이 미미한 실행 구간인 사용자 인지 반응 완료 시점 이후의 구간에 공격적인 저전력 정책을 적용하는 기법을 제안하였다. 또한, 이를 위하여 스마트폰 환경에서 이러한 사용자 인지 반응 완료 시점의 동적 탐색이 가능한 사용자 인지 반응 완료 시간 분석 도구를 개발하였다. 실험 결과에 따르면 제안한 기법은 Android의 기본 CPU 동작 클록 조절 정책인 `ondemand` 대비 최대 65.6% 소비 전력을 개선하는 효과를 보였으며, 이에 따른 사용자가 인지 할 수 있는 성능의 저하는 미미한 수준이었다.

두 번째로, 기존 쓰로틀링 기반의 발열 관리 기법이 사용자 인지 반응 완료 시점 이전 실행 구간에서도 다른 실행 구간과 동일한 방식으로 성능을 제한함에 따라 발생하는 사용자 경험 저하 문제를 연구의 동기로 한 새로운 발열 관리 기법을 제안한다. 기존의 기법은 현재 온도만을 고려하여 쓰로틀링 적용을 결정하는 반면에 제안한 기법은 사용자 인지 현재 실행 구간에서의 시스템 성능이 사용자 경험에 미치는 영향 정도에 대한 정보를 바탕으로 동작하며, 이러한 정보는 반응 완료 시간 분석 도구에 의해 제공된다. 즉, 현재 온도를 바탕으로 결정한 각 쓰로틀링 적용 시점에 쓰로틀링으로 인한 성능 저하가 사용자 경험에 미치는 정도와 해당 시점에서 쓰로틀링을 적용하지 않아 발생하는 온도 상승 정도를 종합적으로 고려함으로써 사용자가 인지할 수 있는 성능은 최대한 보장함과 동시에 시스템에 치명적인 손상을 주는 온도에는 도달하지 않도록 하는 것이다. 실험 결과에 따르면 제안한 기법은 Android에 탑재된 기존의 쓰로틀링 기반의 발열 관리 기법 대비 최대 37.96%의 사용자 인지 반응 완료 시간 개선이 있었다.

세 번째로, 스마트폰 사용자의 개인화된 특성을 다양한 시스템 설계 요구사항 최적화에 사용할 수 있도록 하는 개인화된 최적화 프레임워크를 제안한다.

제안한 개인화된 최적화 프레임워크의 목표는 스마트폰 사용자의 응용 프로그램 사용 기록을 수집하여 누적하고, 이에 대한 분석을 통하여 사용자별로 정형화된 패턴과 경향을 탐색하는 것이다. 사용자별로 응용 프로그램을 사용하는데 나타나는 패턴과 경향의 효율적인 탐색 방법론으로써 사용자의 응용 프로그램 사용 모델을 개발하였다. 또한, 개발한 개인화된 스마트폰 응용 프로그램 사용 모델을 활용하여 응용 프로그램의 재시작을 최대한 방지함으로써 사용자 경험을 향상시킬 수 있는 응용 프로그램 시작 경험 최적화 기법을 제안한다. 실험 결과에 의하면, 제안한 응용 프로그램 시작 경험 최적화 기법이 Android에 탑재된 기존의 LRU 기반의 기법에 대비하여 응용 프로그램 재시작 횟수를 최대 78.4% 감소하는 것을 확인할 수 있었다.

본 논문에서 제안한 기법들은 모두 모바일 OS로 Android를 사용하는 스마트폰 개발 보드 환경에 구현되었으며, 다양한 스마트폰 대상 응용 프로그램을 활용하여 그 효과를 검증하였다. 이를 통하여, 우리는 제안된 사용자 중심의 최적화 기법들이 스마트폰의 소비 전력, 발열, 반응성 등의 다양한 설계 요구 조건들을 최적화하는데 큰 효과가 있음을 확인할 수 있었다.

키워드: 모바일 시스템, 스마트폰, 모바일 운영체제, 시스템 소프트웨어, 사용자 중심 최적화, 사용자 경험, 소비 전력, 발열 관리
학번: 2009-30920