SmartLMK: A Memory Reclamation Scheme for Improving User-Perceived App Launch Time

SANG-HOON KIM, KAIST
JINKYU JEONG and JIN-SOO KIM, Sungkyunkwan University
SEUNGRYOUL MAENG, KAIST

As the mobile computing environment evolves, users demand high-quality apps and better user experience. Consequently, memory demand in mobile devices has soared. Device manufacturers have fulfilled the demand by equipping devices with more RAM. However, such a hardware approach is only a temporary solution and does not scale well in the resource-constrained mobile environment.

Meanwhile, mobile systems adopt a new app life cycle and a memory reclamation scheme tailored for the life cycle. When a user leaves an app, the app is not terminated but cached in memory as long as there is enough free memory. If the free memory gets low, a victim app is terminated and the associated memory to the app is reclaimed. This process-level approach has worked well in the mobile environment. However, user experience can be impaired severely because the victim selection policy does not consider the user experience.

In this article, we propose a novel memory reclamation scheme called *SmartLMK*. SmartLMK minimizes the impact of the process-level reclamation on user experience. The worthiness to keep an app in memory is modeled by means of user-perceived app launch time and app usage statistics. The memory footprint and impending memory demand are estimated from the history of the memory usage. Using these values and memory models, SmartLMK picks up the least valuable apps and terminates them at once. Our evaluation on a real Android-based smartphone shows that SmartLMK efficiently distinguishes the valuable apps among cached apps and keeps those valuable apps in memory. As a result, the user-perceived app launch time can be improved by up to 13.2%.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Main memory*; D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction*

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Android, memory management, mobile device, smartphone

ACM Reference Format:
Sang-Hoon Kim, Jinkyu Jeong, Jin-Soo Kim, and Seungryoul Maeng. 2016. SmartLMK: A memory reclamation scheme for improving user-perceived app launch time. ACM Trans. Embed. Comput. Syst. 15, 3, Article 47 (May 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2894755

1. INTRODUCTION

There have been drastic changes in the computing environment over the last decade. The PC-centric computing environment has been shifted to a mobile computing

This work was supported by the National Research Foundation of Korea (NRF) grant founded by the Korea Government (MSIP) (No. 2013R1A2A1A01016441).
Authors' addresses: S.-H. Kim and S. Maeng, School of Computing, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea; emails: {sanghoon, maeng}@calab.kaist.ac.kr; J. Jeong and J.-S. Kim, College of Information and Communication Engineering, Sungkyunkwan University, 2066 Seobu-ro, Jangan-gu, Suwon 16419, Republic of Korea; emails: {jinkyu, jinsookim}@skku.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 1539-9087/2016/05-ART47 $15.00
DOI: http://dx.doi.org/10.1145/2894755

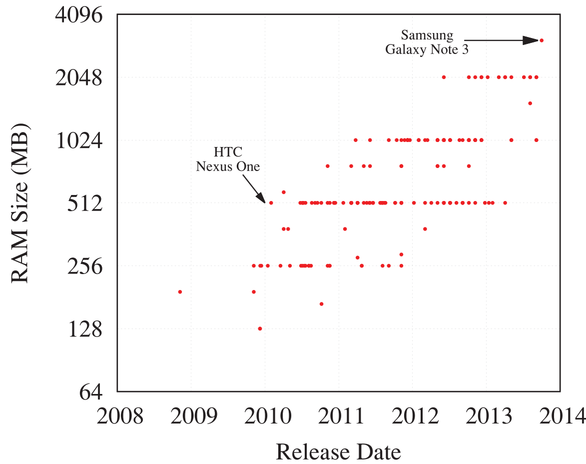

Fig. 1. Equipped RAM size of Android devices over the last 5 years. The y-axis is in log-scale.

environment, in which people use “smart” devices—such as smartphones, smart TVs, smart watches, and tablets—more often than PCs for communication and entertainment. As a result, the yearly sales of smartphones and tablets overtook the sales of PCs in 2011 [Gartner 2011], and 45% of Internet web browsing traffic comes from smartphones and tablets nowadays [Meeker and Wu 2013]. Such a trend is known as “the post-PC era.”

As the post-PC era continues, the mobile computing environment is evolving quickly and the demand for memory has arisen. To satisfy customers’ demands of high-quality apps and better user experience, apps utilize more memory for large buffers and high-quality textures. The increased size and resolution of displays accelerate the memory demand further. To fulfill this ever-increasing memory demand, device manufacturers have adopted the hardware approach, equipping devices with more RAM. Figure 1 shows the quick increment of the equipped RAM size of Android devices over the last 5 years [Wikipedia 2013], and we can see that the equipped RAM size is almost doubled every year. For instance, HTC Nexus One, the first Android reference smartphone released in 2010, is equipped with 512MB of RAM, while the Samsung Galaxy Note 3, released in mid-2013, is equipped with 3GB of RAM. However, such a hardware approach is not a sustainable solution to fulfill the high memory demand. The larger RAM module the device has, the more power the module consumes. This is critical for mobile devices because the usable power provided by batteries is highly constrained and has improved at a slow pace. In addition, the increasing unit cost of devices burdens the device manufacturers. Therefore, efficient management of the limited memory is becoming a challenging issue for mobile devices.

Many studies report the huge difference in app usage patterns between the traditional environment and the mobile environment [Hackborn 2010; Falaki et al. 2010; Rahmati and Zhong 2009; Rahmati et al. 2011; Shepard et al. 2011]. In the mobile environment, users tend to interact with many apps repeatedly and briefly throughout the day. For such a usage pattern, the traditional approach that requires users to close apps explicitly is burdensome. Based on this observation, mobile systems have adopted a new app life cycle model and a process-level memory reclamation scheme tailored for the life cycle. When a user leaves an app, the app is paused and cached in memory rather than being terminated as long as the system has enough free memory. If the amount of free memory drops below a threshold, the process-level memory reclamation

scheme chooses a victim app and reclaims the associated memory by terminating the app. The app can be restored to the last state by saving the last execution state on persistent storage and loading the data when the app is restarted. This approach has worked well for a while.

However, as apps are getting larger and more complicated, apps cannot be recovered to the last state quickly. The design assumption that considers that the apps are lightweight and can be restarted in a short time does not hold anymore [Hackborn 2010], and users have to wait a significant amount of time for apps to be restarted. We face examples of this every day; social networking apps take a few seconds to refresh the up-to-date contents, and 3D game apps load large compressed 3D textures from storage, which can take tens of seconds.

What makes the situation worse is the immature victim selection policy. The policy ignores the influence of the memory reclamation scheme on the user experience and misses many optimization opportunities. The policy prefers the apps with a large memory footprint as the victim. However, such apps tend to take a long time to be restarted. Moreover, the victim app is selected on an on-demand basis, which cannot guarantee the optimality of the victim selection. As a result, the user experience is severely impaired by frequent and long app restart.

In this article, we propose a novel memory reclamation scheme called SmartLMK. SmartLMK collects the characteristics of apps, which can influence the user experience. Based on the data, SmartLMK estimates a temporal penalty that the user will experience if the victim app is terminated. To reclaim memory, SmartLMK selects a set of victim apps by considering the penalty and impending memory demand of the system and terminates them at once. In this way, SmartLMK minimizes the impact of memory reclamation on the user experience. To quantify the impact, we examine a number of ways to capture the user-perceived app launch time and present an effective method to capture the moment when users perceive the app is completely launched.

We implement the proposed scheme on a real Android-based device and evaluate the working prototype with a benchmark with realistic workloads. Evaluation results show that SmartLMK identifies the valuable apps that will impair the user experience severely if they are killed and maintains the valuable apps in memory so that the user-perceived app launch time is improved by up to 13.2%.

The rest of this article is organized as follows. Section 2 goes over the background and explains our motivation. Section 3 proposes a metric to measure the user-perceived app launch time accurately. Section 4 describes the model to estimate the value and memory footprint of apps and the algorithm to select the best victim apps. Section 5 explains our prototype implementation, and Section 6 evaluates the proposed scheme. Section 7 explains related work. Finally, we conclude the article in Section 8.

2. BACKGROUND AND MOTIVATION

2.1. App Life Cycle and Memory Reclamation Scheme in Mobile Devices

There have been a number of software platforms for mobile devices [The Linux Foundation 2013; Apple 2013; LG Electronics 2013; Android 2013a]. Even though they differ in the details of their design and implementation, they attempt to deal with the characteristics of the mobile devices, such as mobility, network connectivity, resource constraints, and so forth, while providing users with full range of exciting user experiences in common. In this article, we will just focus on Android because all the platforms adopt the similar app life cycle and memory reclamation scheme that we will discuss later.

Android [Android 2013a] is one of the most popular platforms in the smartphone market [Gartner 2012]. It is known that 500 million Android devices have been

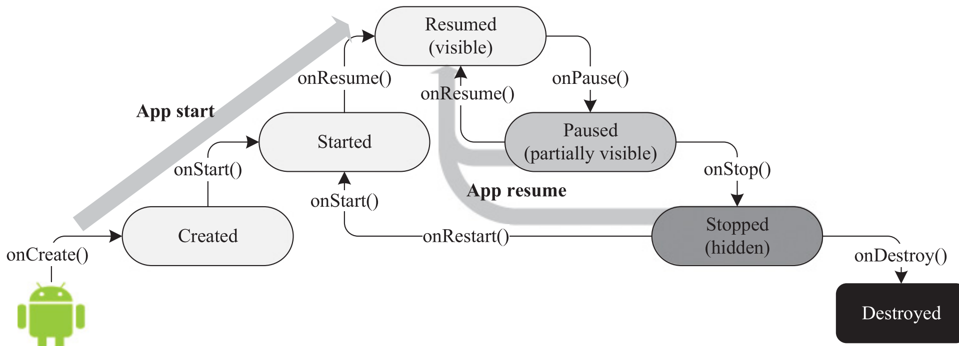

Fig. 2. The life cycle of an Android app.

activated globally, and 1.3 million devices are added every single day as of September 2012 [Barra 2012]. In a big picture, Android is composed of two parts—the Android framework and the Linux kernel. The framework includes runtime, libraries, and application framework and provides apps with platform-specific features such as Dalvik virtual machine, Activity Manager, Package Manager, and so forth. The kernel manages system-level resources such as CPU, memory, and peripheral devices and provides the framework with access to the resources.

Android apps have a unique life cycle that is completely different from that of the desktop and server environment. Figure 2 illustrates the simplified life cycle of Android apps [Android 2013c]. When a user tries to access an app for the first time, the app starts its life cycle from the left bottom state. The Android framework sets up a running environment for the app and invokes callback functions, `onCreate()`, `onStart()`, and `onResume()`, of the app in turn. Along with the callback invocations, the app is gradually changed to `Created`, `Started`, and `Resumed` states. We will refer to this app launch case as *"app start."* When the user leaves the app, the framework invokes the callback functions `onPause()` and `onStop()` and steps down the state of the app to `Paused` and `Stopped` states, respectively. If the user accesses the app again that is in one of these states, the state of the app is changed to the `Resumed` state along with the callback functions `onRestart()` and `onResume()`. We will refer to this app launch case as *"app resume."* We will use the term *"app launch"* to indicate both the app start and the app resume.

Live apps whose states belong to one of `Created`, `Started`, `Resumed`, `Paused`, and `Stopped` are hosted by processes and managed in the *app stack* in the framework. The current foreground app is located at the top of the app stack. If the user tries to use one of the live apps, the foreground app is paused and the target app is brought up to the top of the stack. When the app is in the `Resumed` state, the contents of the newly launched app are drawn on the screen.

The process to host an app is forked from a special process called *Zygote*. Zygote is created during boot time and maps commonly used libraries into the process's address space. While forking the child process from Zygote, the address space of Zygote is copied to the process. In this way, the processes for apps do not have to map the libraries individually so that the mapping cost of frequently used libraries is minimized and the efficiency of memory sharing between processes is maximized.

The memory reclamation scheme in Android is tailored for the life cycle. The live apps are kept in memory as long as the user does not terminate them explicitly and there is enough free memory. When the amount of free memory gets below a threshold, the memory reclamation module called *Low Memory Killer (LMK)* chooses a victim

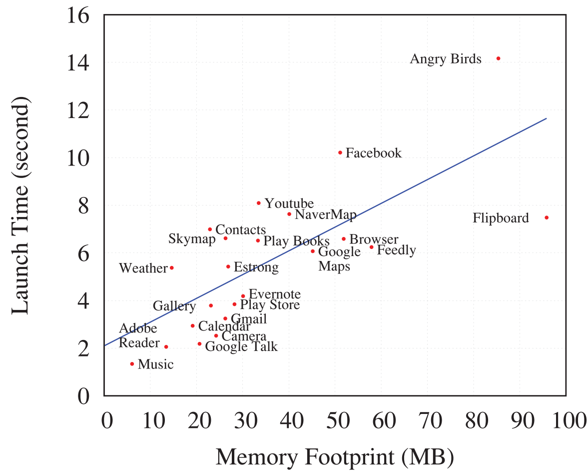

Fig. 3. Memory footprint and launch time of 22 popular Android apps.

app and sends a signal to the process hosting the victim app. Then, the process is terminated and the memory associated with the process is reclaimed while cleaning up the process. The procedure, picking up a victim app and terminating it, iterates until the free memory size reaches the threshold.

As the app termination due to the memory reclamation can happen at any time, apps are recommended to store their execution state in persistent storage. Usually, apps store their execution state when they are being paused or stopped. When the user tries to launch the terminated app, the app can recover to the last state using the stored data. To facilitate the save and recovery of the execution state, the Android framework provides apps with `Bundle` class and the persistent storage such as filesystems and the SQLite database. In this way, Android achieves simple yet effective memory reclamation.

When the app stack is updated due to the app start or resume, the framework classifies the processes hosting live apps into 18 classes. The processes for the system apps and important services, such as telephony and networking services, are classified into the top-ranked classes (-16, -12), and they are not considered as the victim for the memory reclamation. The midranked classes (0–8) are composed of the processes for the current foreground app, visible apps (e.g., widget and wallpaper apps), user-perceptible apps (e.g., music player), the home app, backup apps, and service apps. The other live apps are classified into the low-ranked classes (9–15) where the more recently used app is assigned to the higher-rank class. The classification results are passed down to the Linux kernel through the /proc filesystem and stored in the per-process data structure `task_struct`. When memory gets low, the LMK picks the victim app that belongs to the lowest class and occupies the largest amount of memory.

2.2. Motivation

As we described in the previous subsection, Android employs the new app life cycle and the process-level reclamation scheme tailored for mobile devices. However, the victim selection policy of the scheme is incomplete and has three aspects to be optimized in terms of user experience.

First, the policy prefers the apps with a large memory footprint to those with a small footprint. This preference, however, can harm the user experience. Figure 3 shows the relationship between the memory footprint and launch time of 22 popular

Android apps, measured on the Google Galaxy Nexus running Android Ice Cream Sandwich (ICS) 4.0.4. The launch time means the time to initialize the app and to display up-to-date contents. The line represents the linear regression between the memory footprint and the launch time and shows the correlation coefficient of $r = 0.74$. From the strong correlation, we can infer that the apps with a large memory footprint tend to take a long time to be restarted. Therefore, selecting large apps as the victim for the memory reclamation can have a negative impact on user experience because if they are killed, users may need to wait for the long restart time.

Second, the policy does not consider the app's launch time at all and misses the opportunity to provide better user-perceived app launch time. Even apps with a similar memory footprint can exhibit different launch times. In Figure 3, Facebook and Browser have almost the same memory footprint, approximately 50MB, but they differ in launch time by 3.6 seconds. If they have the same likelihood of being launched later, reclaiming memory from Browser interrupts users less, yet the same amount of memory can be reclaimed. However, the current policy does not utilize the feature and could aggravate the user experience.

Lastly, as the reclamation happens on an on-demand basis, if a number of apps need to be killed to secure a large amount of memory, the apps are not killed together but one after another. This is problematic because the on-demand approach can mislead the decision to a suboptimal sequence of app termination and cannot reclaim memory in a concurrent way. Consider three apps whose memory footprint is 40, 30, and 20MB, and their launch time is 4, 3, and 1 second, respectively. If an app demands 50MB of memory, the policy will pick up the first two apps in turn because they have the largest memory footprint when the decision is made. It will take 7 seconds to restart the killed apps later. However, the memory demand can be fulfilled at a lower cost by killing the latter two apps; this requires only 4 seconds for restoring the killed apps. This example suggests that the victim selection policy can be optimized by considering the memory demand and taking a proactive approach.

From these three aspects, we can conclude that the current memory reclamation scheme can be refined by considering the user experience. Our approach is to assess apps in terms of the impact on user experience if the app is terminated and to select victim apps so that the impact on the user experience due to the app termination is minimized. We quantify the impact on the user experience from the perspective of the *user-perceived app launch time*. However, measuring the user-perceived app launch time is a challenging issue because it is hard to know when users notice that an app is launched. We will describe how our scheme measures the user-perceived app launch time and how we utilize the metric in the memory reclamation scheme.

3. MEASURING USER-PERCEIVED APP LAUNCH TIME

The app launch time is an important performance metric for mobile devices [Apple Computer 2006]. However, it is difficult to clearly tell when the app is completely launched. As a result, previous studies employ ad hoc metrics [Joo et al. 2011; Kim et al. 2012] or use the app launch time provided by the Android framework as their performance metric [Jeong et al. 2012, 2013a, 2013b]. But those metrics cannot deal with complex app launch scenarios or reflect the actual user-perceived app launch time.

In this section, we examine a number of ways to measure the app launch time and analyze the plausibility of them. We also introduce an efficient way to capture the user-perceived app launch time.

3.1. Previously Used Metrics

In the Android-based systems, it is obvious that an app launch starts when the user interacts with the system to launch the app. However, it is unclear when the app is

completely launched because there is no explicit definition about the moment when the app is completely launched. The moment can vary from different points of view. We can consider the following moments as the time when the app launch is completed.

—**First Visible.** An app launch is completed when the app becomes visible on the screen. The rationale for this metric is that users react to change on the screen and believe that the app becomes visible because it is ready. The Android framework provides the launch time based on this way.
—**First Interactive.** An app launch is completed when the user can see an interactive screen so that the app responds to user interaction immediately. The reason for considering this metric is that users perceive that the app launch is in progress if the app does not respond to user interaction.
—**Up-to-Date.** An app launch is completed when the app becomes visible and its contents are refreshed up-to-date. The rationale for this metric is that users will not consider the app as completely launched until they can access the latest contents that they actually intend to access.
—**App Decides.** Apps decide themselves when they are completely launched. This metric is reasonable because the app itself knows the best time when it is completely launched.

Those four moments will not differ much if apps are simple. However, apps are getting complicated nowadays and may take tens of seconds to be launched. To make users feel that the long launch is shorter, apps employ a number of psychological techniques such as splash screen, load progress screen, and delayed initialization [Apple Computer 2006], which improve the user-perceived performance by satisfying humans with feedback. These techniques affect the examined moments and make the situation more complicated. For example, First Visible is simple but can conclude the app launch prematurely. It cannot capture the user-perceived launch time if at least one of the psychological techniques is used—users will not consider the app to be fully launched while they see the splash screen. First Interactive and Up-to-Date may handle these psychological techniques properly, but there is no obvious and autonomous way to determine when the screen becomes interactive or when the contents become up-to-date. App Decides may provide the most accurate methodology to determine the app launch time. However, every app needs to be changed and recompiled, and malicious app developers may abuse the scheme by reporting the moment of completion in a wrong way.

From these observations, we need to find a new and effective way to capture the user-perceived app launch time without considerable modification of the framework and/or apps.

3.2. Tracking App Launch

Our approach is to infer the launch completion by means of monitoring system resource usage. An app launch accompanies the use of system resources such as CPU, storage, and networking—the process to host the app needs to be forked from Zygote, and the app binary needs to be fetched from the storage. Also, the associated app data may be read or downloaded, decompressed, and rendered on the screen. All of these activities utilize many system resources. When the app is completely launched, the app will wait for a user interaction, which requires fewer resources than the busy phase during the app launch. Based on those characteristics, we propose an effective way to capture the moment of app launch completion by tracking the system resource usage.

Every app launch initiates a new tracking session, and the session ends when the app is considered to be completely launched. During the session, system resource usage is measured at an interval I. The current system resource usage U is measured

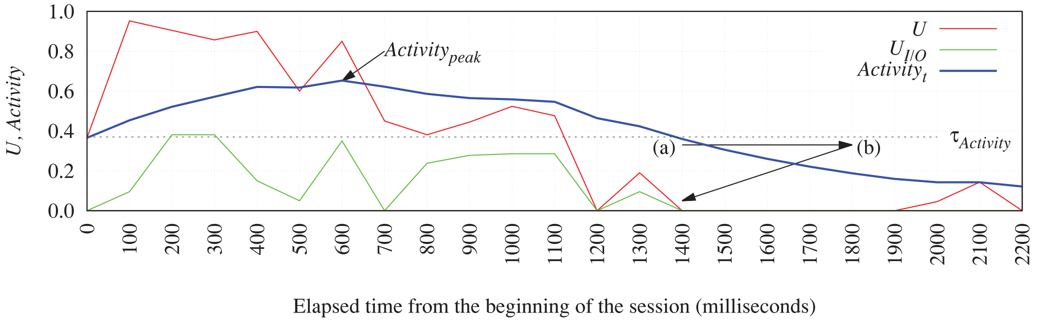

Fig. 4. An example of the launch tracking.

by the portion of the ticks spent in the busy or I/O wait state over the total elapsed ticks since the last measurement and can be expressed as

$$U = \frac{\text{ticks in busy} + \text{ticks for I/O wait}}{\text{total elapsed ticks}}. \tag{1}$$

U is 0 when the system is idle and 1 when the system is fully utilized. We include the I/O wait ticks in U to take the I/O utilization and the CPU utilization into account.

U does not include past history of the resource utilization and fluctuates greatly even if the app launch is in progress. To address the limitations, we model the resource use activity $Activity_t$ by means of the exponential moving average of U as follows:

$$Activity_0 = U_0 \tag{2}$$

$$Activity_t = \alpha \cdot Activity_{t-1} + (1 - \alpha) \cdot U_t, \tag{3}$$

where t is the sequence number of the measurement and U_0 is the U when the launch tracking session is started. The α is a tunable weight between 0 and 1 and controls how much the past activity affects the current resource use activity. The greater α is, the more the past history affects the current activity.

When the app is completely launched, the resource activity decreases and it remains at a low level for a while until the user begins to interact with the app. To capture this characteristic, we employ two parameters, τ_{peak} and τ_N. The first parameter, τ_{peak}, determines whether the resource use activity is decreased enough. A threshold $\tau_{Activity}$ is derived from the peak resource use activity within the session, $Activity_{peak}$, multiplied by τ_{peak} such that $0 < \tau_{peak} \leq 1$:

$$\tau_{Activity} = Activity_{peak} \times \tau_{peak}. \tag{4}$$

τ_N determines whether the decreased resource use activity has been kept for a certain amount of time. If the current activity $Activity_t$ is lower than or equal to the threshold $\tau_{Activity}$ during τ_N intervals, we consider that the resource usage is sufficiently decreased and stabilized. So, we end the tracking session and conclude when the app launch is finished. If the total elapsed time for the tracking session is T, the actual app launch time T_{launch} is given by

$$T_{launch} = T - I \times \tau_N, \tag{5}$$

where I is the interval to measure the resource utilization.

Figure 4 depicts an example of the launch tracking proposed in this article. In the example, we use $\tau_{peak} = 0.55$, $\tau_N = 5$, and $I = 100$ milliseconds. Label (a) points to the time when $Activity_t$ gets below $\tau_{Activity}$ for the first time, and Label (b) stands for the end of the session. In this case, the app launch time is estimated to 1,400ms.

Table I. Parameters for Tracking App Launch

Parameter	Description	Value Used
I	Interval to sample the system resource utilization	100 ms
α	Parameter to smooth out temporary utilization change	0.90
τ_{peak}	Threshold to determine whether the app becomes idle	0.56
τ_N	Threshold to determine how long the app has been idle	17

Table I summarizes the introduced parameters and their values we will use to the rest of the article. We will explain how to obtain these parameter values in Section 6.2. Also, it is noteworthy that our launch tracking technique does not distinguish the app start and the app resume and can be applied to both cases seamlessly.

4. MEMORY RECLAMATION SCHEME OPTIMIZATION

As we gave an example in Section 2.2, the memory reclamation scheme in Android does not consider the impact of app termination on user-perceived launch time and misses many opportunities to improve the user experience. In addition, the on-demand approach that selects one victim app at a time can mislead the victim selection to a suboptimal sequence of app termination.

In this section, we describe how we address the problems and optimize the memory reclamation scheme. We transform the original victim selection problem to the 0/1 knapsack problem and build mathematical models to assess the value and weight of apps with the user-perceived app launch time, app use preferences, and the history of memory usage. By solving the transformed version of the original problem, we can select a set of victim apps so that the impact of their termination on the user-perceived app launch time is minimized while the impending memory demand of the system is quickly satisfied.

4.1. Memory Reclamation and 0/1 Knapsack Problem

The goal of the victim selection policy of the memory reclamation in Android is to keep the most valuable apps in memory while the sum of their memory footprint does not exceed the memory constraint. For given n apps $a_1, a_2, \ldots, a_n$, where the value and memory footprint of each app are denoted as Value(a_i) and Memory(a_i), respectively, we can formulate the goal as follows:

$$\text{Maximize} \sum_{i=1}^{n} \text{Value}(a_i) \cdot x_i \text{ subject to } \sum_{i=1}^{n} \text{Memory}(a_i) \cdot x_i \leq \text{Available memory}, \tag{6}$$

where $x_i \in \{0, 1\}$. This optimization is known as the 0/1 knapsack problem [Kellerer et al. 2004], which maximizes the sum of the value while the sum of the memory footprint does not exceed the given amount of available memory. If we know Value(a_i), Memory(a_i), and the maximum allowed memory of the system, we can put them in the formula and obtain the optimal solution [Kellerer et al. 2004]. From the solution, we can select victim apps to terminate by collecting the apps a_i such that $x_i = 0$.

In the following, we will describe how we model Value(a_i), Memory(a_i), and the maximum memory footprint of the system. Then, we will summarize how we obtain the optimal solution from the models.

4.2. App Value Modeling

Previous studies [Falaki et al. 2010; Rahmati et al. 2011; Shepard et al. 2011] suggest that the apps installed on a device are not equally used and do not behave in the same way. Some apps take tens of seconds to be launched, whereas some apps are launched almost instantly. Some apps are used on a daily basis, whereas some apps are not used

at all. Due to the diversity among the apps and the variety of factors that influence the diversity, it is not easy to answer how, and also in terms of what we can evaluate the value of an app.

As mobile apps are highly interactive and users are sensitive to the app launch time [Apple Computer 2006; Joo et al. 2011], we estimate the value of an app a_i, $Value(a_i)$, considering the user-perceived app launch time and the likelihood of being used. Recall that apps have different start time and resume time, and the apps cached in memory can be launched via the app resume, which generally takes a shorter time than the app start. Assume that the app start time and the app resume of the app a_i are $T_{start}(a_i)$ and $T_{resume}(a_i)$, respectively. If the app is cached, the next app launch will be completed in $T_{resume}(a_i)$ via the app resume. If the app is killed, the next app launch will be served through the app start that takes $T_{start}(a_i)$. Hence, if the app a_i is killed, the user will experience the prolonged user-perceived app launch time $\Delta T(a_i)$ on the next use of the app a_i:

$$\Delta T(a_i) = T_{start}(a_i) - T_{resume}(a_i). \tag{7}$$

We can extend this model to the case with n cached apps $a_1, a_2, \ldots, a_n$. If the apps have the same likelihood of being launched next and an app a_i is selected as the victim, then the *expected* increment of the user-perceived app launch time will be $\Delta T(a_i)/n$. However, apps are not equally preferred, and there are temporal and spatial localities in app usage [Falaki et al. 2010; Shepard et al. 2011]. As apps have a different likelihood of being used by the next app launch, each app will have different contributions to the user-perceived app launch time if the app is killed. We need to take this difference into account.

For an app a_i, we model the likelihood of the use of the app as $P(a_i)$. However, $P(a_i)$ is highly dependent on the app usage pattern, and predicting $P(a_i)$ accurately is a challenging issue due to the immense diversity of the app usage. For example, Falaki et al. [2010] try to capture the distribution of app usage and suggest that the temporal length of an app's use can be modeled by an exponential distribution. Other studies [Yan et al. 2012; Zhang et al. 2013; Parate et al. 2013] suggest statistical models to predict the apps that are most likely to be used soon. However, we cannot adopt these models since they utilize the contexts of an app launch (e.g., location and time), which is not available to our environment.

For this reason, we devise a model for our own purposes. We assume that picking an app to launch among apps with varying popularity is an independent random event. Then, the app selection is a Poisson process [Hayter 2012], which implies the reuse distance of an app (i.e., the number of app launches between two launches of the app plus one) follows an exponential distribution. Note that the reuse distance of two consecutive launches is defined as 1 for brevity of the equations. Specifically, when an app has an average reuse distance r, the actual reuse distance obeys an exponential distribution whose cumulative distribution function $F(d;r)$ is defined for the reuse distance $d \geq 1$ as follows:

$$F(d;r) = 1 - e^{-d/r}. \tag{8}$$

Since the exponential distribution is memoryless, the probability to pick the app for the next app launch is $F(1;r)$. Thus, we can derive $P(a_i)$ as

$$P(a_i) = F(1;r_i) = 1 - e^{-1/r_i}, \tag{9}$$

where r_i stands for the average reuse distance of a_i. Note that we do not claim that the model outperforms other models but provide a reasonable statistical model for predicting the probability of an app launch. We believe that a more sophisticated approach can improve the app value estimation by improving the accuracy of the prediction.

Finally, we can obtain the value of an app a, $Value(a)$, as follows by referring to Equations (7) and (9):

$$\begin{aligned} Value(a_i) &= \text{The influence of the app's termination} \times \text{the likelihood of being used} \\ &= \Delta T(a_i) \times P(a_i) \\ &= \big(T_{start}(a_i) - T_{resume}(a_i)\big) \times (1 - e^{-1/r_i}). \end{aligned} \quad (10)$$

4.3. App Memory Modeling

It is required that the memory footprint of apps, Memory(a_i), and the total available memory, Available memory, be estimated in Equation (6). However, it is not obvious because various memory-sharing techniques such as page sharing, file mapped pages, and kernel same-page merging are all blended [Kim et al. 2014].

There are a number of metrics to represent the memory footprint of a process. The virtual set size (VSS) is the size of memory region the process is allocated. The allocation happens when the process reserves a portion of its process address space. However, as the actual page allocation is delayed until the address is referenced, the VSS greatly overestimates the actual memory footprint of the process. The resident set size (RSS) is the primary metric for the memory footprint in the Linux kernel and is obtained from the number of pages that are mapped into the process's address space. When a page is mapped into/unmapped from the process's address space, the RSS is increased/decreased by the size of the page. Because RSS takes the actually used pages into account, RSS is much more accurate than VSS. However, RSS has no information on shared pages so that the memory footprint can be misled if a large portion of memory is shared with other processes. Recall that every app shares a portion of its address space with Zygote in Android as we described in Section 2.1. Therefore, RSS is not a good measure in the Android environment.

To address the limitations of these metrics, Mackall [2007] introduces two new metrics and kernel patches. The proportional set size (PSS) counts the degree of memory sharing in accounting for the memory footprint. If n processes share a page, the page size divided by n is counted for each process's PSS. The unique set size (USS) counts the pages that are not shared by any other processes. As a page will be reclaimed by the system when no process maps the page anymore, the USS is a very accurate metric for estimating the size of the reclaimable memory when the process is killed. However, obtaining the PSS and USS is very costly because it requires one to walk through the page table of the process and to examine the sharing degree of each virtual memory area. Thus, it is infeasible to use them directly in memory reclamation logic.

Instead, we propose an effective and accurate way to estimate the reclaimable memory size of a process in the Android environment. Since apps are forked from Zygote, the pages mapped to Zygote's address space are also mapped to the app's address space and the shared memory regions cannot be reclaimed by killing the app. The rest of the memory can be reclaimed unless it is shared with other processes. Android provides a number of ways to facilitate data sharing between apps. The sharing is usually done by means of IPC via the Android binder, and direct memory sharing between processes is rare. Therefore, we can anticipate that the memory that is not shared with Zygote is not shared with other processes as well and the very amount of memory can be reclaimed by killing the app. Therefore, we can obtain $Memory(a_i)$, the effective memory footprint of app a_i, as follows:

$$Memory(a_i) = RSS(a_i) - RSS(\text{Zygote}). \quad (11)$$

Note that the RSS of Zygote is determined during the system boot time and remains steady. Therefore, we can obtain $Memory(a_i)$ efficiently without traversing the page table and virtual memory area.

The apps running in a system utilize the memory by allocating or deallocating it. If an app a_i is about to allocate memory by $Alloc(a_i)$ at a moment, then the current memory demand of the system *Demand* will be given as follows:

$$Demand = \sum_{i=1}^{n} Alloc(a_i) - Free, \tag{12}$$

where *Free* denotes the free memory in the system. As the paused or stopped apps are dormant so that they incur little memory activity, we can regard the memory allocation from these apps to be negligible. For the active app, we assume that the memory usage is self-similar and estimate the additional memory allocation at a moment from the history of the memory footprint of the app as follows:

$$Alloc(a_i) = RSS_{peak}(a_i) - RSS(a_i), \tag{13}$$

where $RSS_{peak}(a_i)$ is the maximum memory footprint of the app in the past. Thus, Equation (12) can be reduced to

$$Demand = RSS_{peak}(a_{top}) - RSS(a_{top}) - Free, \tag{14}$$

where a_{top} is the foreground app that is on top of the app stack.

4.4. Optimal Victim Selection

With the value model and the memory model, the goal expressed in Equation (6) can be rewritten as the following 0/1 knapsack problem:

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^{n} Value(a_i) \cdot x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} Memory(a_i) \cdot x_i \leq \sum_{i=1}^{n} Memory(a_i) - Demand, \end{aligned} \tag{15}$$

where $x_i \in \{0, 1\}$. By solving the problem, we can obtain the optimal victim apps V by collecting the apps such that $V = \{a_i | x_i = 0\}$. By killing the victim apps, we can reclaim memory by $\sum_{a \in V} Memory(a)$, which is greater than or equal to the additional memory demand *Demand*. Note that the current Android victim selection policy picks up one victim app at a time, whereas our approach does a set of victim apps at once. By reclaiming the memory in bulk, our approach can boost the reclamation speed.

The 0/1 knapsack problem is known to be NP-complete and can be solved in pseudo-polynomial time and space, $O(n \cdot Demand)$, with the dynamic programming technique [Kellerer et al. 2004]. We can trade off the complexity and the granularity of the solution by quantizing the memory footprint to a desired granularity.

5. IMPLEMENTATION

We have implemented a working prototype of the proposed scheme on a Google Galaxy Nexus smartphone, which runs the Android framework 4.3_r3 Jelly Beans (JB) on the Linux kernel 3.0.8. We have implemented four modules in the system—Launch Tracker, History Store, SmartLMK, and Knapsack Solver—and Figure 5 outlines these modules and their interactions.

To launch an app, the Activity Manager Service, the core system service in the Android framework, sets up the running environment for the app and notifies the

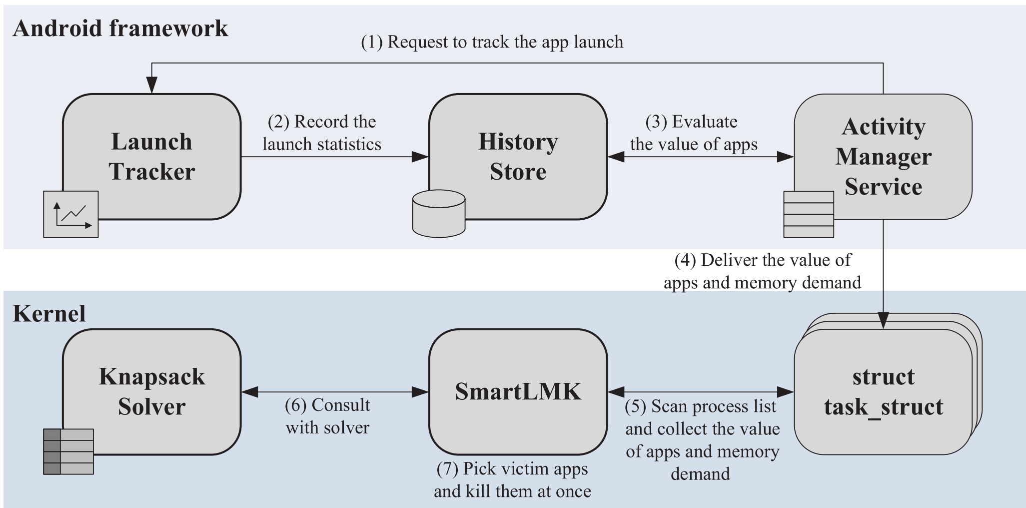

Fig. 5. The overall architecture of SmartLMK.

Launch Tracker to track the app launch. The Launch Tracker tracks the app launch as we described in Section 3. It starts a new tracking session and monitors the system resource usage. When the app launch is completed, the Launch Tracker ends the tracking session and stores the measured launch time and memory footprint in the History Store.

The History Store provides persistent storage for data that are used to estimate the value and memory demand of apps. The data include the number of the app launch, type of app launch (start or resume), app start time, app resume time, maximum RSS, and sum of the reuse distance. The data pertaining to app usage are collected when the app is launched. The data pertaining to the memory footprint are measured when the app becomes paused and maintained with the exponential moving average with the decaying factor $\alpha = 0.85$.

The Activity Manager Service updates the class of app processes when the app stack is altered. We modified the Activity Manager Service to assess the value of apps and the current memory demand as we described in Sections 4.2 and 4.3, respectively, during the app class update. The required data are fetched from the History Store. The assessed results are handed over to the Linux kernel via the `/proc` filesystem interface and stored in the `task_struct` of each process.

When the free memory of the system gets below a threshold, SmartLMK collects the values, memory footprints, and memory demand and consults the Knapsack Solver to get optimal victim apps. The Knapsack Solver solves the 0/1 knapsack problem that we described in Section 4.4 and provides SmartLMK with a list of victim apps. We have implemented the Knapsack Solver using dynamic programming. Note that the Knapsack Solver requires a working space to solve the problem, but the system is already low on free memory when the Knapsack Solver is running. Therefore, allocating additional memory at this moment may fail. To avoid this situation, the working space of the Knapsack Solver is preallocated during system boot-up.

As SmartLMK obtains the list of victim apps from the Knapsack Solver, it kills them at once by sending a signal to the corresponding processes. In this way, SmartLMK reclaims memory in bulk from the multiple victim apps.

Table II. Apps and Their Categories in the Benchmark Workload

Category	App	Launch	Fraction (%)
Communication	Gmail	24	25.8
	SMS	24	
	Phone	18	
Game	Candy Crush Saga	18	18.0
	Angrybirds Star Wars	14	
	Subway Surfer	14	
Social networking	Facebook	18	14.8
	Flipboard	16	
	Feedly	4	
Multimedia	Camera	12	10.9
	Youtube	12	
	Gallery	4	
Productivity	Evernote	16	9.4
	Adobe Reader	4	
	Calendar	4	
Browser	Browser	20	7.8
Navigation	Google Maps	12	4.7
News	ESPN ScoreCenter	8	4.7
	Weather	4	
Shopping	Play Store	10	3.9
Total	20 apps	256	100.0

6. EVALUATION

6.1. Experimental Setup

We evaluated the proposed scheme with the prototype implemented on a Google Galaxy Nexus smartphone. It is equipped with TI OMAP 4460 SoC (1.2GHz dual-core ARM Cortex-A9 CPU), 1GB of RAM, and 16GB of eMMC [Google 2011]. A server connected to the smartphone via USB controls the device with Android Debug Bridge (ADB) and Monkeyrunner [Android 2013b, 2013d].

To evaluate the performance of the scheme on a realistic workload, we composed a benchmark that imitates a user. The benchmark iterates to launch an app, wait for the app to be completely launched, idle for 10 seconds after the launch completion, launch the home app, and idle for another 5 seconds. The sequence of the app to launch is described in a workload file. We generated the workload with 20 popular apps from various categories in Play Store. We carefully assigned the ratio of the apps and categories by referring to the research on user behaviors [Falaki et al. 2010; Shepard et al. 2011] to make the workload realistic. Table II summarizes the composition of the benchmark workload. The sequence is randomly generated once considering the composition ratio and used throughout the evaluation.

While we evaluated our system, we observed performance variations that stem from various system components. To minimize the effect from the system components, we carefully set up the environment as follows:

—**Background activity.** Each measurement starts with rebooting the system and waits 3 minutes for system activities to subside. Then, all background processes are terminated using console command "`am kill-all`." Also, we turn off the automatic synchronization and update features from apps.
—**Page cache.** We equalize the condition of the page cache across measurements by dropping all page cache pages before starting the benchmark.

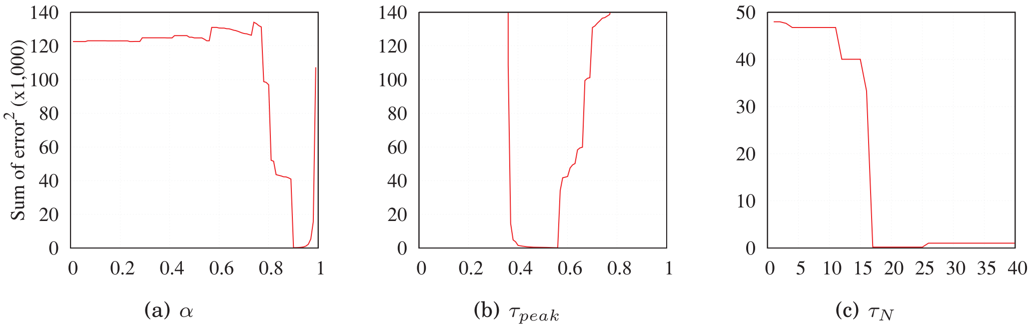

Fig. 6. The sensitivity of the parameter value for tracking app launch.

—**CPU frequency governor.** CPU frequency governors can dynamically change CPU frequency and result in performance variations. We use the *"Performance"* governor so that cores statically run at the highest available CPU frequency.
—**Thermal throttling.** The SoC of the device throttles clock frequency down if cores get overheated. We observed that throttling happens occasionally and influences the measurement. We remove the back cover from the device and place a cooling fan behind the device to avoid the unwanted throttling. This eliminates the thermal throttling throughout the evaluation.
—**Networking.** All Wi-Fi channels in the 2.4GHz band are very crowded at the measuring site, and channel collision influences the stability of the network performance. We use a vacant 5GHz band to avoid the channel collision.

6.2. App Launch Tracking Validation

First, we describe how we pick the parameters for the app launch tracking, which are listed in Table I. We empirically derived the values from the actual resource utilization of popular apps. First, we collected a resource usage trace at an interval while launching the apps on the evaluation device. We picked 100 milliseconds as the interval I since the interval provides a decent time resolution for measuring the app launch time ranging from hundreds of milliseconds to tens of seconds. While collecting the samples, we obtained the actual user-perceived app launch time of each launch by recording the screen of the device on a video and analyzing the video frame by frame. As the video has 24 frames per second, we can obtain 42ms of temporal resolution. Then, we estimated the app launch time of the sample launches using every possible combination of the parameter values. We picked the parameter values that minimize the sum of the square of the estimation errors (i.e., the difference between the actual app launch time and the estimated one).

Using these values that minimize the sum of the square errors, we analyze the sensitivity of the parameter values. We measured the sum of the square errors while varying one parameter value. The remaining two parameter values are fixed to the values shown in Table I. Figure 6 summarizes the result. α and τ_{peak} have a range of values that make the error considerably small. When α is too small, the launch tracking becomes so sensitive to the current resource utilization that temporal resource usage variance in the middle of the app launch can mislead the tracking. On the other hand, too large an α can mislead the tracking as well by making the tracking reflect the past too much. τ_{peak} shows a similar pattern. If τ_{peak} is too small, any resource use after a launch prevents the tracking from being finished. If the value is too large, the tracking can be prematurely finished by temporal resource usage variance. Unlike these two

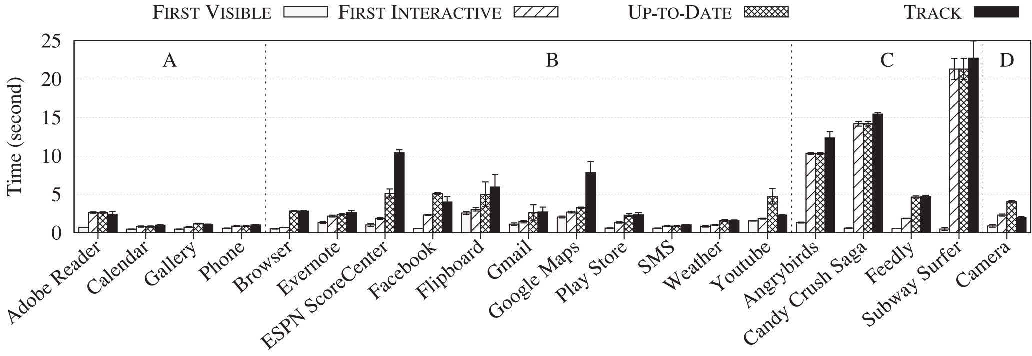

Fig. 7. App start time measured by various metrics.

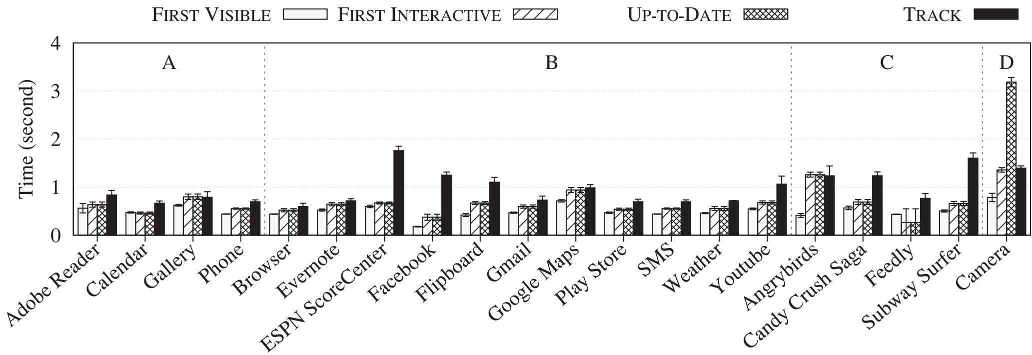

Fig. 8. App resume time measured by various metrics.

parameters, τ_N is not very sensitive to large values. However, when τ_N is too small, the launch tracking can be misled by the resource usage variance.

Now, we validate how accurately the proposed launch tracking scheme measures the user-perceived app launch time in practice with these parameter values. We investigated each app and identified the moments when an app is first shown, when it becomes interactive, and when the screen becomes up-to-date, each of which corresponds to First Visible, First Interactive, and Up-to-Date in Section 3, respectively. We determined the moments from each app by observing the app's response to user inputs while it is being launched. For example, the Facebook app becomes visible with a flat white screen and becomes interactive with user inputs when its UI components are shown. The screen becomes up-to-date when the news feed is drawn. And then we recorded a video of the target device's screen and ADB consoles simultaneously while running a script that starts or resumes an app repeatedly. App start is performed by killing the process that hosts the app prior to launching the app, whereas app resume is carried out by launching the app again without killing the process. We analyzed the recorded video frame by frame and obtained the app launch times according to the metrics by measuring the elapsed times when the screen is changed to the corresponding moments. We assume that users consider that an app is completely launched when the app finishes rendering up-to-date contents on the screen. Hence, we regard Up-to-Date as the actual app launch time users perceive. Note that Track refers to the launch tracking method we propose.

Figures 7 and 8 summarize the app start time and the app resume time obtained from the frame-by-frame analysis, respectively. The time is an average of eight runs

and the error bar represents the standard deviation of the time. We classify the apps into four groups according to their resource usage behavior during their app launch. The labels A to D above the bars indicate the group. Apps within a group are listed in alphabetical order from left to right.

Group A represents simple apps that show the current contents on the first visible screen and do not update the contents later. Even for these simple apps in this group, FIRST VISIBLE cannot accurately measure the user-perceived launch time for their app start. When an app is launched through the app start, it needs to read, parse, and render contents, which requires a considerable amount of time. However, FIRST VISIBLE does not count the time to process contents. Consequently, there are gaps between the actual user-perceived app launch time and the measured time for these apps. Contrary to the app start, FIRST VISIBLE can capture the user-perceived launch time well for app resumes because the app resume does not require much time to process contents. In this group, TRACK also captures the actual app launch time well.

Group B includes networked apps that display an interactive screen first, checks updated contents in background, and refreshes the contents on the screen up-to-date. A user can start interacting with the apps after FIRST INTERACTIVE but should wait until the updated contents appear at UP-TO-DATE. In this group, FIRST VISIBLE concludes the app launch prematurely and fails to capture the actual moment. The launch time measured by FIRST VISIBLE differs from the actual user-perceived launch time by up to 2,558ms. Compared to FIRST VISIBLE, TRACK is closer to the actual app launch time except for ESPN ScoreCenter, Facebook, Flipboard, and Google Maps. For these apps, TRACK largely overestimates their app start time (ESPN ScoreCenter and Google Maps) and the app resume time (ESPN ScoreCenter, Facebook, and Flipboard). An in-depth analysis finds that the phenomenon comes from the background activities of the apps that are captured by TRACK but have no visual update. When the apps are started, ESPN ScoreCenter downloads news stories and Google Maps processes maps in the background after updating contents on the screen. ESPN ScoreCenter, Facebook, and Flipboard check for an update when they are resumed. These works are not visible to a user unless there is an interaction or an update for the contents during the app launch. However, there was not an interaction nor an update during the evaluation, making TRACK overestimate their app launch time. We verified that the actual app launch time (i.e., UP-TO-DATE) is increased accordingly if there is an interaction or an update is involved during their launch so that TRACK becomes close to the actual user-perceived app launch time for the apps.

Group C includes the apps that show a splash screen and the progress of initialization when they are started. In this case, a user can play the game or see the contents after finishing the initialization. FIRST VISIBLE is far from the actual app start time because it concludes the app start when the splash screen appears. However, TRACK can measure the app start much more accurately so that the error is much smaller than that of FIRST VISIBLE. For the app resume, TRACK measures the app resume time longer than the actual time due to the background activity or additional resource usage. However, we can say that TRACK is more robust than FIRST VISIBLE given that the worst-case error of TRACK is much smaller than that of FIRST VISIBLE from the app start.

Group D is composed of the Camera app where TRACK coincides with FIRST INTERACTIVE. The app utilizes the system resource that is monitored by TRACK while initializing the app. However, after the initialization, the app switches to use a hardware accelerator (e.g., GPU) for emulating a viewfinder on the screen. For this reason, TRACK cannot capture the actual app launch time for Camera.

To summarize, the proposed method for tracking app launch can measure the actual user-perceived app launch time better than other metrics. For the rest of the article, we will use the launch time measured by the proposed scheme.

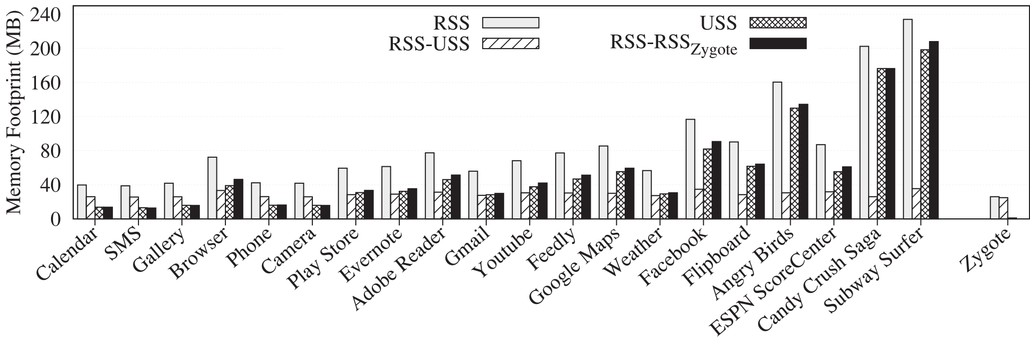

Fig. 9. The memory footprint measured by various metrics.

6.3. Memory Model Validation

We verify the memory model proposed in Section 4.3. We measure the memory footprint of target apps using `procrank`, which is a supplementary tool in the Android Open Source Project. The measurement is done when the apps become background because the system reclaims memory from background apps rather than foreground apps. Figure 9 shows the memory footprint of each app. Throughout the evaluation, the RSS of Zygote is observed to be 26.0MB.

The RSS of the apps varies from 38.8MB to 234.0MB, but the shared portion of memory (i.e., RSS−USS) is almost static across the apps. Specifically, the average of the shared portion is 29.3MB, which is close to the RSS of Zygote. The USS of an app is close to the RSS of the app subtracted by the RSS of Zygote. Also, we observe that killing an app increases the free memory by the USS of the app. Therefore, USS, the reclaimable memory size of an app, can be effectively and accurately estimated with the RSS of the app and the RSS of Zygote.

6.4. Performance of SmartLMK

We evaluate the performance of SmartLMK using the benchmark we discussed in Section 6.1. We compare the performance of SmartLMK with the baseline memory management policy of the stock firmware, which we will refer to as “Vanilla.” To further analyze the implications of system memory configurations, we deploy a kernel module that is similar to the balloon driver in the ESX server [Waldspurger 2002]. It limits system memory by preoccupying the designated amount of memory. Therefore, a larger balloon size implies less memory that the system can utilize and higher memory pressure.

To quantify the influence of SmartLMK on the user experience, we measured the total time spent launching apps while running the benchmark. A longer app launch time implies a longer user-perceived app launch time and a worse user experience since a user has to wait longer to access apps. Figure 10 summarizes the results from various memory configurations.

We can observe that the total app launch time increases along with memory pressure. Specifically, the total time increases from 1,619.6 seconds to 1,872.2 seconds for Vanilla, and from 1,404.5 seconds to 1,739.5 seconds for SmartLMK. We attribute the increase of the total time to the decreased number of cached apps. The system can cache fewer apps in the presence of memory pressure. Fewer cached apps results in a lower probability that an app to launch is cached in memory. Consequently, more app launches are done via app start, which usually takes a longer time than app resume, thereby increasing the total app launch time. This is illustrated in Figure 11, which presents the number

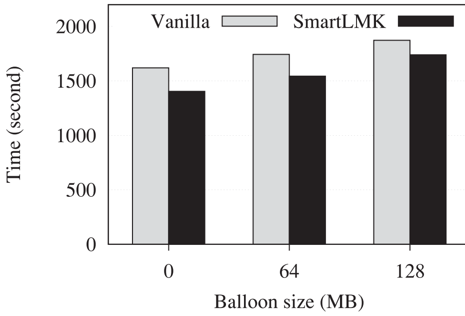

Fig. 10. Total time spent launching apps.

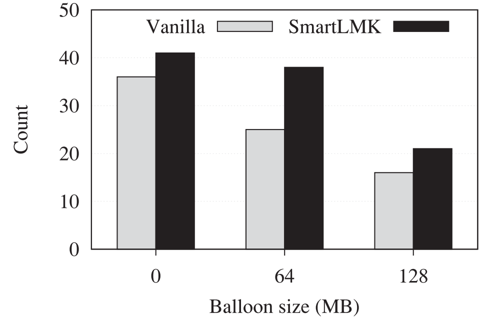

Fig. 11. The number of app resumes out of 256 app launches.

of app resumes out of 256 app launches in the benchmark. In both SmartLMK and Vanilla, the number of app resumes decreases as memory pressure is increased.

Across all memory configurations, SmartLMK outperforms Vanilla. Specifically, SmartLMK improves the total user-perceived app launch time by 13.2% when the balloon size is 0, which indicates that SmartLMK can effectively improve the user experience. The amount of improvement (i.e., the time gap between SmartLMK and Vanilla) decreases from 215.1 to 132.7 seconds as memory pressure is increased from 0MB to 128MB. The decrease originates from the reduced number of cached apps when memory pressure is applied. If many apps are cached, SmartLMK has many chances to optimize by exploring their combinations. However, SmartLMK cannot optimize much with few apps in hand. As a result, SmartLMK cannot improve the total app launch time much in these smaller memory configurations. Despite the reduced chances, SmartLMK can optimize with the apps in hand and improves the total app launch time by 7.1% when the balloon induces 128MB of memory pressure.

The evaluation result also shows that the app usage prediction of SmartLMK, described in Section 4.2, outperforms the original LRU scheme. As shown in Figure 11, the total app resume count of SmartLMK is larger than that of Vanilla in all memory configurations. Besides, an analysis reveals that SmartLMK maintains fewer apps in memory than Vanilla since SmartLMK proactively reclaims memory; when the balloon size is 0, the average number of cached apps during the benchmark is measured to be 6.70 and 5.94 for Vanilla and SmartLMK, respectively, and the same trend is observed from other memory configurations as well. These results indicate that SmartLMK derives more app resumes from fewer cached apps. In this regard, we can conclude that the proposed app usage prediction scheme identifies the apps that are likely to be used again more accurately than the original scheme of Vanilla.

To further analyze the factors that improve the total app launch time in SmartLMK, we present the average app launch time of each app in Figure 12 and the breakdown of the app launches by type in Figure 13. For simplicity, we only present the results from the memory configuration without memory pressure since the same pattern is observed from other memory configurations. The apps are arranged in increasing order of their average launch time in Vanilla from left to right.

We can notice the trend that the app resume counts for the apps with a short launch time (ranging from Calendar to Adobe Reader) are decreased, whereas the counts for the apps with a long launch time (ranging from Gmail to Subway Surfer) are increased mostly. These changes come from the optimization strategy of SmartLMK, which keeps the apps that are valuable in terms of user experience. Recall that the value of an app is estimated with $T(a_i)$, the app launch time prolonged by converting app resume

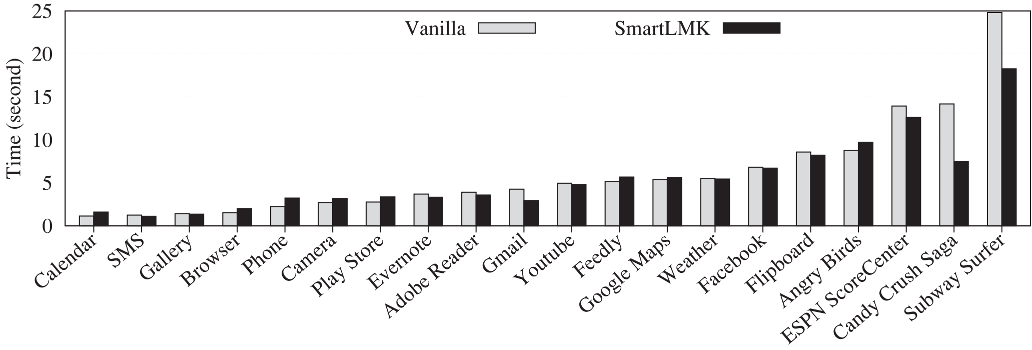

Fig. 12. The average user-perceived app launch time of each app.

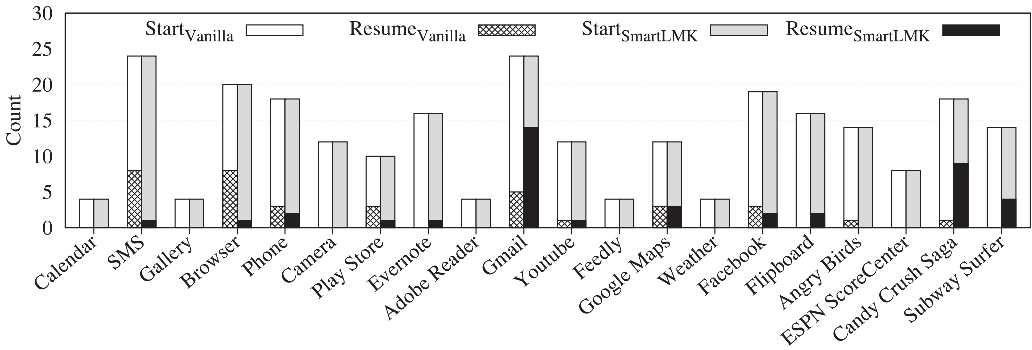

Fig. 13. The breakdown of app launches by types.

to app start, and $P(a_i)$, the likelihood that the app is being used, as described in Equation (10).

The apps ranging from Calendar to Adobe Reader have an app start time that does not differ much from the app resume time as shown in Figures 7 and 8. Thus, their $T(a_i)$ values are small, and converting app resume to app start does not extend their average app launch time much. SmartLMK reclaims these apps preferably, and their app resume counts are decreased by seven for SMS and Browser, and by two for Play Store. Contrary to these apps, the apps ranging from Gmail to Subway Surfer have a considerably longer app start time than app resume time. Thus, these apps have large $T(a_i)$ values, which implies that their average app launch time is heavily influenced by their app resume count. For this reason, SmartLMK tries to keep these apps in memory, and the app resume count is increased by nine, two, eight, and four for Gmail, Flipboard, Candy Crush Saga, and Subway Surfer, respectively. Consequently, the average app launch time is reduced by up to 6.7 seconds for Candy Crush Saga, which is 47.1% of the average launch time of the app in Vanilla.

Comparing SMS and Gmail shows the implications of SmartLMK as well. We can assume they are equal in terms of recency and frequency since they have the same app launch counts in the workload. However, the original victim selection policy of Vanilla prefers Gmail to SMS since Gmail has a larger memory footprint than SMS. Thus, Gmail is more likely to be killed during memory reclamation, making its app resume count smaller than that of SMS. In contrast, SmartLMK prefers SMS to Gmail since SMS extends the average app launch time less than Gmail when it is terminated. As a

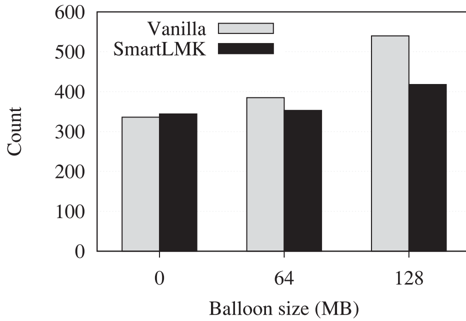

Fig. 14. The number of process kills while running the benchmark.

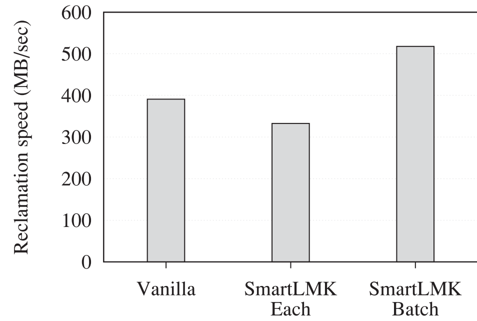

Fig. 15. Memory reclamation speed.

result, the app resume count of Gmail is significantly increased, whereas the count of SMS is notably reduced in SmartLMK.

6.5. Influence on Memory Reclamation

To analyze the influences of SmartLMK on memory reclamation, we counted the number of processes killed by LMK while running the benchmark. The result is summarized in Figure 14. When the balloon size is zero, fewer processes are killed in Vanilla than SmartLMK. However, the number of kills quickly increases along with the memory pressure so that it outnumbers that of SmartLMK when the balloon size exceeds 64MB. This trend is caused by empty apps and the on-demand memory reclamation strategy of Vanilla. The empty app means the process hosting a content provider [Android 2014] without a running or pending request. The content provider can provide data and related services to other apps. When an app attempts to acquire data from a content provider, the Android framework spawns a process and executes the content provider within the process. The spawned process is not terminated after serving the request to accelerate subsequent requests to the content provider. Instead, the process is cached as the *empty app* and reclaimed later via LMK like normal cached apps. In the evaluation benchmark, a number of apps acquire data from Google Service Framework, which is the content provider of the contacts, appointment schedules, and backup facilities for a registered Google account. Thus, launching these apps causes empty apps to be spawned. In Vanilla, LMK is triggered when the free memory gets below thresholds. Since the memory footprints of empty apps are usually very small, LMK cannot reclaim much memory from these small empty apps, and terminating an empty app is usually followed by another memory reclamation. However, the empty app is respawned when an app requires data from the content provider. Therefore, empty apps are frequently spawned and terminated, and LMK is frequently invoked in Vanilla. In contrast, LMK is less frequently triggered in SmartLMK since SmartLMK reclaims memory proactively by terminating multiple processes simultaneously, and empty apps are reclaimed along with normal cached apps if they are found. Thus, empty apps are less frequently killed, and the number of process kills is reduced.

To compare the memory reclamation performance, we measured the memory reclamation speed. The speed is obtained from the sum of the memory footprint of each victim app divided by the total time spent to reclaim the memory. Figure 15 shows the memory reclamation speed when the balloon size is zero. The results from other memory configurations are omitted for brevity since the same trend is observed from the configurations. In SmartLMK, multiple processes can be simultaneously reclaimed in a memory reclamation batch, and their reclamation time may overlap each other.

In this sense, "SmartLMK Each" indicates the reclamation speed for each process kill, whereas "SmartLMK Batch" is obtained by considering a memory reclamation batch as a whole. Note that SmartLMK Batch is the memory reclamation speed that the system actually experiences.

As shown in Figure 15, Vanilla outperforms SmartLMK Each by 17.6%, whereas SmartLMK Batch outperforms Vanilla by 32.5%. An additional analysis indicates that SmartLMK picks 1.67 processes per memory reclamation batch on average. The concurrent process termination slows down individual memory reclamation since the app terminations in a batch compete with each other for resources and can be serialized at a microscopic level. In contrast, the concurrent memory reclamation scheme of SmartLMK utilizes parallelism in memory reclamation and boost memory reclamation performance even though individual memory reclamation is slowed down.

6.6. Overhead of SmartLMK

SmartLMK is composed of Launch Tracker, History Store, and Knapsack Solver as we described in Section 5. Launch Tracker is a thread in the Android framework that is idle when an app launch is not in progress. Thus, Launch Tracker does not incur any overhead for most of time. When an app is being launched, Launch Tracker wakes up and tracks the launch. The tracking is to read data from the kernel, and it takes only 231 microseconds on average in our evaluation device. Thus, we can assume the overhead of Launch Tracker is insignificant.

History Store collects per-app statistics in the system. The statistics include the total number of app launches for each app launch type (start or resume), the sum of the app start times, the sum of the app resume times, the maximum RSS of the app, and the sum of the reuse distances. Thus, History Store uses 56 bytes (seven long words) per app, and the total space overhead is proportional to the number of installed apps in the system.

Knapsack Solver requires a 2D array of integers as a working space to solve the 0/1 knapsack problem. The width of the array is determined by the maximum number of apps that the solver can deal with at a time. We used eight as the maximum number of apps since Android caches up to eight apps in the background. The height of the array is bounded to the maximum memory size the system can utilize for running apps. The size can be quantized to a desired granularity to bound the time and space complexity of the solver. In our evaluation device, the system can utilize approximately 500MB of RAM for running apps since the rest of RAM is reserved for hardware or occupied by system processes that cannot be reclaimed. In this sense, we set the maximum memory size to 512MB. In addition, we quantize the memory footprint to 1MB granularity since the memory footprint of apps ranges from a few megabytes to hundreds of megabytes. Hence, the working space of the solver takes 16KB ($8 \times 512 \times 4$) of memory.

The time complexity to solve a problem is proportional to the number of apps and the memory demand, and the time is measured to take up to 800 microseconds in the evaluation device. As the solver is invoked approximately 200 times while running the benchmark, we can attribute that 160 milliseconds to the time overhead came from the solver. The overhead is negligible compared to the total benchmark running time and the time improved by SmartLMK.

7. RELATED WORK

There have been several discussions to understand and analyze the characteristics of the mobile computing environment [Eagle and Pentland 2006; Falaki et al. 2010; Rahmati et al. 2011; Rahmati and Zhong 2009; Shepard et al. 2011; Trestian et al. 2009]. Falaki et al. [2010] present a comprehensive study of smartphone use. They collect detailed usage traces from 255 Android and Windows Mobile platform users,

which span 7 to 28 weeks per user. Along all aspects of their study, including app uses and preferences, users differ by one or more orders of magnitude. The immense diversity suggests that if systems learn and adapt to user behaviors, the user experience can be much improved. They also find similarity among users as well as the diversity.

Livelab studies [Shepard et al. 2011; Rahmati et al. 2011] conduct long-term field research on smartphone usage. They collect an 18-month length of traces from 22 carefully selected iPhone 3GS users. The traces comprise app launch, charging, phone call, app preference, and so on. They find that the use of apps changes over the year, highlight the influence of socioeconomic status on device usage, and reveal how some users are more similar to each other than others. Their traces are publicly available on the Internet [Rice Efficient Computing Group 2012].

Many studies have investigated how user contexts can be utilized in a computing environment [Bishop 2007; Hudson et al. 2003; Ravi et al. 2005]. Chu et al. [2011] point out that smartphones are equipped with many sensors—microphone, GPS, gyroscope, accelerometer, barometer, digital compass, proximity sensor, and camera—and the sensors can offer opportunities for context-aware computing. They examine a number of feasible context-aware OS services and design a framework to handle the user contexts in OSs. Yan et al. [2012] provide a prototype of the context-aware OS service called FALCON. FALCON models user interactions with smartphones and identifies user preferences among installed apps from various contexts. Based on the preferences, FALCON predicts candidate apps that fit the best to the current user context and preloads the apps so that the user can access them instantly.

Utilizing memory efficiently is the traditional research issue from many systems and has been studied for a long time. Compcache and ZRAM [Gupta 2010; Magenheimer 2013] increase the effective memory size by compressing rarely used pages. While swapping out a page, the page is compressed and packed efficiently in a RAM-based virtual block device if the page is compressed well. Kernel Same-page Merging (KSM) [Arcangeli et al. 2009] introduces the content deduplication to the memory system. A thread periodically scans the memory and removes duplicated pages at the cost of CPU. Transcendent Memory (Tmem) [Magenheimer et al. 2009; Oracle 2013] improves the memory utilization by adding an abstraction layer in the memory system. Tmem maintains a memory pool by claiming underutilized memory from the system. The pool can be utilized for swap pages (Frontswap), page cache (Cleancache), and so forth.

These “page-level” approaches can complement the process-level scheme described in Section 2.1. As these page-level approaches are transparent to user-level applications, the applications do not have to be altered. Also, they can manage the memory in a finer granularity than the process-level scheme. However, the advantages of the page-level schemes can be easily offset by the process-level scheme because user applications can be terminated and restored frequently—the effort to manage the memory of the terminated applications returns little benefit but the management overhead.

8. CONCLUSION

We have identified the problems of the current process-level reclamation scheme. The victim selection policy does not consider the impact of app termination on the user experience, and misses optimization opportunities. To address these problems, we have quantified the influence of app termination on the user experience by means of the user-perceived app launch time. We have transformed the original victim selection problem to the 0/1 knapsack problem and obtained the optimal victim apps by modeling the value and weight of apps. Our comprehensive evaluation shows that the proposed scheme, SmartLMK, improves the overall user-perceived app launch time by up to 13.2%.

We plan to develop the value and memory model of apps further and to extend our approach to other aspects such as energy, storage traffic, and network traffic. We also anticipate that the main idea of our approach can also be applied to mobile systems and platforms other than Android.

REFERENCES

Android. 2013a. Homepage. Retrieved from http://www.android.com.

Android. 2013b. Android Debug Bridge (ADB). Retrieved from http://developer.android.com/tools/help/adb.html.

Android. 2013c. Managing the Activity Lifecycle. Retrieved from http://developer.android.com/training/basics/activity-lifecycle/index.html.

Android. 2013d. Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html. (September 2013).

Android. 2014. Content Providers. Retrieved from http://developer.android.com/guide/topics/providers/content-providers.html.

Apple Computer. 2006. Launch Time Performance Guidelines. Retrieved from http://developer.apple.com/documentation/Performance/Conceptual/LaunchTime/LaunchTime.pdf.

Apple. 2013. Apple iOS 7. Retrieved from http://www.apple.com/ios.

Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the 2009 Ottawa Linux Symposium (OLS'09)*. 19–28.

Hugo Barra. 2012. 500 Million. Retrieved from https://plus.google.com/110023707389740934545/posts/R5YdRRyeTHM.

Christopher M. Bishop. 2007. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York.

David Chu, Aman Kansal, Jie Liu, and Feng Zhao. 2011. Mobile apps: It's time to move up to CondOS. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'11)*. 16–20.

Nathan Eagle and Alex Pentland. 2006. Reality mining: Sensing complex social systems. *Personal and Ubiquitous Computing* 10, 4 (2006), 255–268.

Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. 2010. Diversity in smartphone usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*. 179–194.

Gartner. 2011. Sales of Smartphones and Tablets to Exceed PCs. Retrieved from http://www.practicalecommerce.com/articles/3069-Sales-of-Smartphones-and-Tablets-to-Exceed-PCs.

Gartner. 2012. Forecast: Mobile Devices by Open Operating System, Worldwide, 2009-2016, 2Q12 Update. (2012).

Google. 2011. Google Galaxy Nexus. Retrieved from http://www.google.com/nexus/#/tech-specs.

Nitin Gupta. 2010. compcache: Compressed Caching for Linux. Retrieved from http://code.google.com/p/compcache.

Dianne Hackborn. 2010. Multitasking the Android Way. Retrieved from http://android-developers.blogspot.kr/2010/04/multitasking-android-way.html.

Anthony Hayter. 2012. *Probability and Statistics for Engineers and Scientists*. Brooks/Cole, Cengage Learning.

Scott Hudson, James Fogarty, Christopher Atkeson, Daniel Avrahami, Jodi Forlizzi, Sara Kiesler, Johnny Lee, and Jie Yang. 2003. Predicting human interruptibility with sensors: A wizard of oz feasibility study. In *Proceedings of the 2003 SIGCHI Conference on Human Factors in Computing Systems (CHI'03)*. ACM, 257–264.

Jinkyu Jeong, Hwanju Kim, Jeaho Hwang, Joonwon Lee, and Seungryoul Maeng. 2012. DaaC: Device-reserved memory as an eviction-based file cache. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'12)*. 191–200.

Jinkyu Jeong, Hwanju Kim, Jeaho Hwang, Joonwon Lee, and Seungryoul Maeng. 2013a. Rigorous rental memory management for embedded systems. *ACM Transactions on Embedded Computer Systems* 12, 1s, Article 43 (March 2013), 21 pages.

Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013b. I/O stack optimization for smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. USENIX Association, 309–320.

Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. 2011. FAST: Quick application launch on solid-state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (USENIX FAST'11)*. 259–272.

Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. *Knapsack Problems*. Springer-Verlag, Berlin.

Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Conference on File and Stroage Technologies (USENIX FAST'12)*.

Sung-Hoon Kim, Jinkyu Jeong, and Joonwon Lee. 2014. Efficient memory deduplication for mobile smart devices. In *Proceedings of the 2014 IEEE International Conference on Consumer Electronics (ICCE'14)*.

L. G. Electronics. 2013. Open webOS. Retrieved from http://www.openwebosproject.org.

Matt Mackall. 2007. System-wide memory profiling. In *Proceedings of the 2007 CELF Embedded Linux Conference (ELC'07)*.

Dan Magenheimer. 2013. In-Kernel Memory Compression. Retrieved from http://lwn.net/Articles/545244/.

Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. 2009. Transcendent memory and linux. In *Proceedings of the 2009 Ottawa Linux Symposium (OLS'09)*.

Mary Meeker and Liang Wu. 2013. KPCB Internet Trends 2013. (May 2013).

Oracle. 2013. Project: Transcendent Memory. Retrieved from http://oss.oracle.com/projects/tmem.

Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. 2013. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'13)*. 275–284.

Ahmad Rahmati, Clayton Shepard, Chad Tossell, Mian Dong, Zhen Wang, Lin Zhong, and Phillip Kortum. 2011. Tales of 34 iPhone users: How they change and why they are different. *Technical Report TR-2011-0624* (2011).

Ahmad Rahmati and Lin Zhong. 2009. A longitudinal study of non-voice mobile phone usage by teens from an underserved urban community. *Technical Report TR-0515-09* (2009).

Nishkam Ravi, Nikhil Dandekar, Preetham Mysore, and Michael L. Littman. 2005. Activity recognition from accelerometer data. In *Proceedings of the 7th Conference on Innovative Applications of Artificial Intelligence (IAAI'05)*. American Association for Artificial Intelligence, 1541–1546.

Rice Efficient Computing Group. 2012. LiveLab: Measuring wireless networks and smartphone users in the field. http://livelab.recg.rice.edu/traces.html. (July 2012).

Clayton Shepard, Ahmad Rahmati, Chad Tossell, Lin Zhong, and Phillip Kortum. 2011. LiveLab: Measuring wireless networks and smartphone users in the field. *Performance Evaluation Review* 38, 3 (January 2011), 15–20.

The Linux Foundation. 2013. Tizen: An open source, standards-based software platform for multiple device categories. http://www.tizen.org. (2013).

Ionut Trestian, Supranamaya Ranjan, Aleksandar Kuzmanovic, and Antonio Nucci. 2009. Measuring serendipity: Connecting people, locations, and interests in a mobile 3G network. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference (IMC'09)*. 267–279.

Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*.

Wikipedia. 2013. Comparison of Android Devices. http://en.wikipedia.org/wiki/Comparison_of_Android_devices. (2013).

Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services (MobiSys'12)*.

Chunhui Zhang, Xiang Ding, Guanling Chen, Ke Huang, Xiaoxiao Ma, and Bo Yan. 2013. Nihao: A predictive smartphone application launcher. *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering* 110 (2013), 294–313.

Received December 2013; revised November 2015; accepted February 2016