Personalized Optimization for Android Smartphones

WOOK SONG, YESEONG KIM, HAKBONG KIM, JEHUN LIM,
and JIHONG KIM, Seoul National University

As a highly personalized computing device, smartphones present a unique new opportunity for system optimization. For example, it is widely observed that a smartphone user exhibits very regular application usage patterns (although different users are quite different in their usage patterns). User-specific high-level app usage information, when properly managed, can provide valuable hints for optimizing various system design requirements. In this article, we describe the design and implementation of a personalized optimization framework for the Android platform that takes advantage of user's application usage patterns in optimizing the performance of the Android platform. Our optimization framework consists of two main components, the application usage modeling module and the usage model-based optimization module. We have developed two novel application usage models that correctly capture typical smartphone user's application usage patterns. Based on the application usage models, we have implemented an app-launching experience optimization technique which tries to minimize user-perceived delays, extra energy consumption, and state loss when a user launches apps. 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.

Categories and Subject Descriptors: D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Android, application usage pattern, mobile systems, user experience

ACM Reference Format:
Wook Song, Yeseong Kim, Hakbong Kim, Jehun Lim, and Jihong Kim. 2014. Personalized optimization for Android smartphones. ACM Trans. Embedd. Comput. Syst. 13, 2s, Article 60 (January 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2544375.2544380

1. INTRODUCTION

Unlike traditional computing devices, such as PCs, smartphones are truly personalized computing devices. Given that there is only one dominant user per smartphone, collecting and analyzing this single user's usage tendencies provide a new novel opportunity for optimizing various system design requirements, such as user experience, performance, and energy consumption. For example, understanding typical usage behavior of this single user allows a smartphone to make more intelligent operational decisions.

A possibility of this type of *personalized optimization* is supported by various smartphone usage studies. For example, a recent study [Wireless Intelligence 2011] shows

This research was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (No. 2010-0020724). The ICT at Seoul National University and IDEC provided research facilities for this study.
Authors' addresses: W. Song, Y. Kim, H. Kim, J. Lim, and J. Kim (corresponding author), Department of Computer Science and Engineering, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul 151-742, Korea; email: {wooksong, yeseong, haknalgae, brownrabbit, jihong}@davinci.snu.ac.kr.

DOI: http://dx.doi.org/10.1145/2544375.2544380

that only a small number of different apps in the rest of this article, we call an application by a (more popular) shorthand, app are used by each user, although there are more than 200,000 apps available from the Android app market [Digitizor 2011]. 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 for a total of 2,454 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 the next three apps launched following *Memo App*.

If we could identify these distinctive usage patterns among a small set of favored apps during runtime, we could improve, for example, user experience by giving these apps favors. One such example might be when more free memory is necessary. In such situation, these favored apps may be excluded from a list of background apps which might be terminated to provide the needed free memory.

In this article, 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 have 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.

In order to evaluate the effectiveness of the POA framework on proposed launching experience optimization schemes, we have implemented the POA framework in the Android platform, version 2.3 (Gingerbread) running on the Nexus S Android reference smartphone [Google 2010]. Our evaluation results, which were based on real app usage logs collected from active smartphone users, show that our proposed optimization techniques reduce unnecessary app restarts by up to 78.4% compared to the Android's default policy.

The rest of this article is organized as follows. After presenting our analysis results on how smartphone apps are used in Section 2, we describe an overview of the POA framework and illustrate how the POA framework can be utilized to improve the launching experience using a small example in Section 3. We explain two proposed app usage models in Section 4, while the proposed optimization techniques are described in Sections 5 and 6. In Section 7, experimental results are discussed, and Section 8 summarizes related work. Finally, in Section 9, we conclude with a summary.

2. SMARTPHONE APP USAGE ANALYSIS

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 nonintrusive fashion while users interact with their favored apps. This app automatically collects high-level information on smartphone use, including information about the start and end of each app use, the detailed breakdown

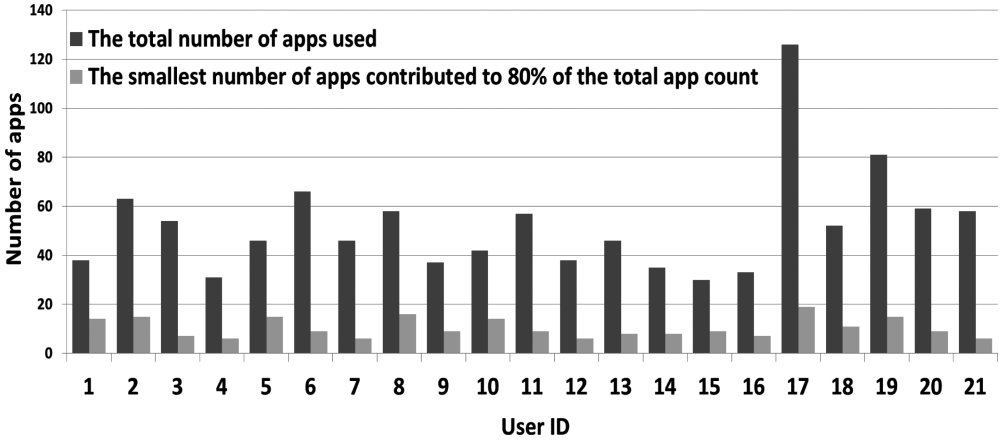


Fig. 1. Per-user app usage distribution.

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

Fig. 2. Per-app frequency as one of three next launched apps right after each *Memo App* launch (total 38 *Memo App* launches).

of how a user interacts 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 *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 1, 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 2 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 2 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.

3. OVERVIEW OF POA

The proposed POA framework consists of two main modules: the app usage modeling module and the app usage model-based optimization module. Figure 3 shows an overview of the POA framework within the Android platform. The app usage modeling module extracts meaningful app usage patterns from execution logs and the app

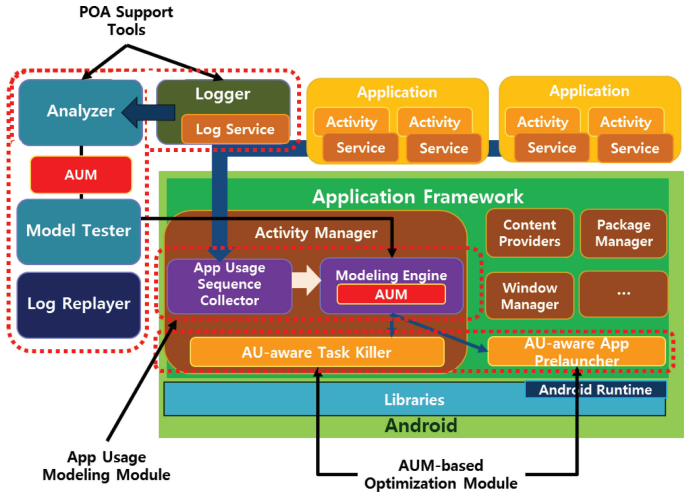


Fig. 3. An architectural overview of the proposed POA framework.

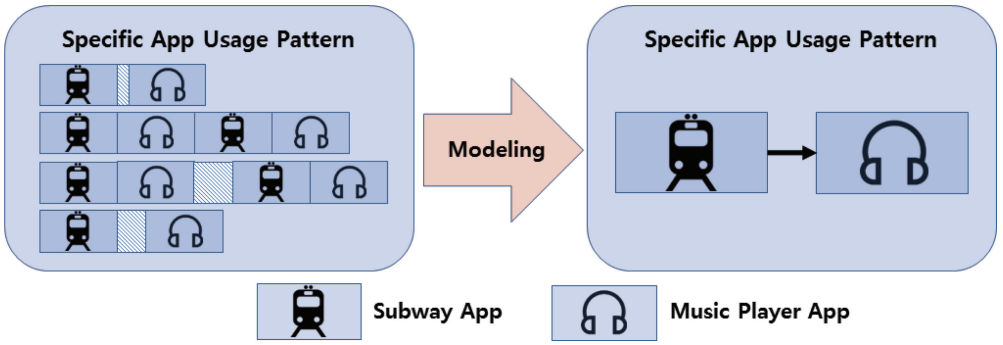
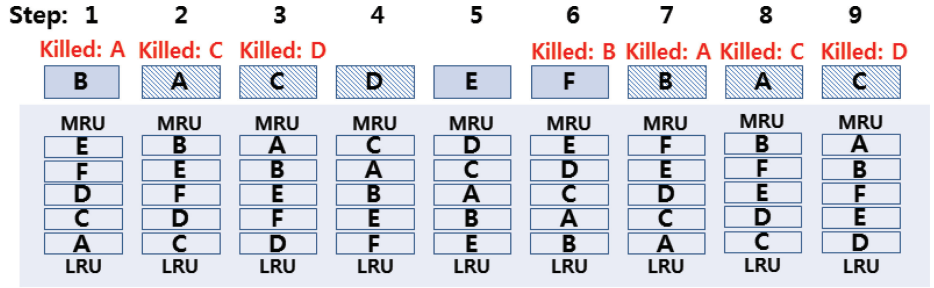


Fig. 4. An example of building an app usage model.

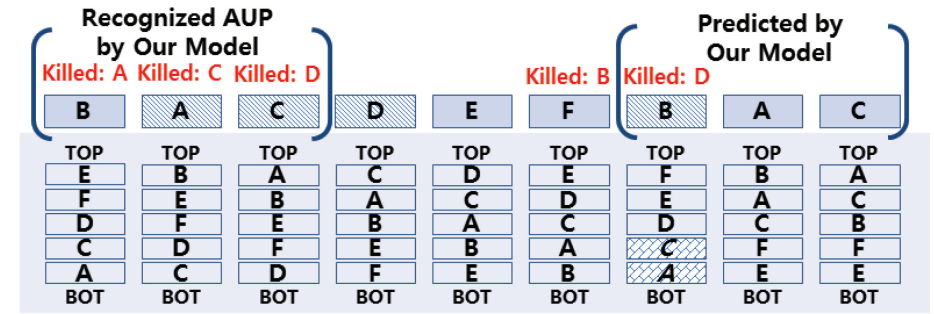
usage model-based optimization module is responsible for exploiting AUMs for various optimizations. In order to efficiently evaluate various optimization schemes developed within the POA, we have built several offline custom POA support tools as well. The logger, which is explained in Section 2, and analyzer tools are used for collecting app usage logs and analyzing them offline, 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 ten seconds so that different solutions can be quickly explored.

Figure 4 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 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 4.

Once the AUM is constructed from the app usage modeling module, we can take advantage of the AUM's knowledge on the user behavior for improving the system performance and user experience. Figure 5 conceptually illustrates how the AUM can be used in improving the LRU-based task killing policy of the Android platform. In



(a) Based on LRU: Restart Counts: 6.



(b) Based on AUM: Restart Counts: 4.

Fig. 5. An example of using an app usage pattern in optimizing the LRU-based task-killing policy.

this example, we assume that the LRU stack contains only five apps for 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 right after B is used. Therefore, D will be killed instead at 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.

3.1. 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 is 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

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

initializes a AUM module at boot time. Then, user's app usage data collected by the app usage sequence collector are passed to the modeling engine.

3.2. Usage Model-Based Optimization Module

The usage model-based optimization module consists of various submodules which are optimization specific. 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 Section 6). 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.

4. 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 $\mathcal{S} = \langle a_1, a_2, \ldots, a_l \rangle$ represent an app usage log, where a_i indicates the ith 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 $\mathbb{D}_\mathcal{S}$ of distinct apps in the app usage log $\mathcal{S}$ as $\mathbb{D}_\mathcal{S} = \{a_{i_1}, \ldots, a_{i_k}\}$, where the set $\mathbb{D}_\mathcal{S}$ consists of distinct apps launched in $\mathcal{S}$ (i.e., 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 $\mathcal{S} = \langle a_1, a_2, a_3, a_4 \rangle = \langle x, y, y, x \rangle$.

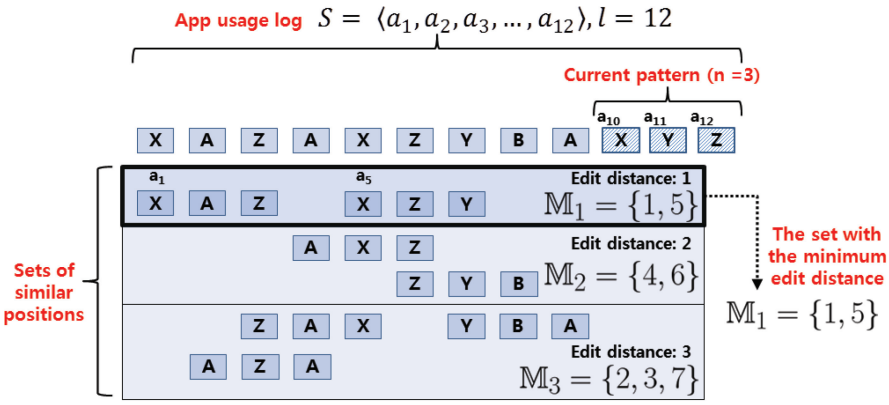
4.1. P-AUM: Pattern-Based App Usage Model

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 $\mathcal{S}$ 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 $\mathcal{S}$. 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 $\mathcal{S} = \langle a_1, a_2, \ldots, a_l \rangle$. 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 [Levenshtein 1966]. We represent a similar position by its index in $\mathcal{S}$.

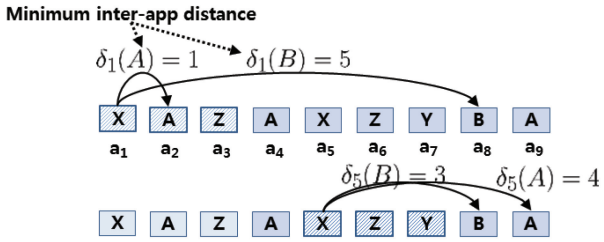
In this algorithm, a similarity metric, called the *edit distance*, is calculated for measuring how similar two strings are. The edit distance represents the minimum number of edits to transform one string into the other by insertion, deletion, substitution, and transposition. In the P-AUM heuristic, 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 6(a) illustrates the modeling process just explained. 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 sequence. For example, in Figure 6(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 *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



(a) Current pattern and similar positions.



(b) Calculating the minimum inter-app distances for each app.

Fig. 6. An example of how the P-AUM heuristic finds apps to be launched next.

minimum inter-app distance $\delta_{s_i}(x)$ for a given $\mathcal{S} = \langle a_1, \ldots, a_l \rangle$ and a similiar 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) $1 \leq s_i \leq j \leq l$ and $x = a_j$.
(ii) There is no t such that $a_t = x$ for $s_i < t < j$.
(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 condition (i) or (ii) is not satisfied), $\delta_{s_i}(x)$ is defined as $|\mathbb{D}_{\mathcal{S}}|$.

For example, in Figure 6(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 6(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 $\mathcal{S} = \langle a_1, a_2, \ldots, a_l \rangle$ 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 $O(|M| \cdot |N|)$ 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 $\mathcal{S} = \{a_1, \ldots, a_{2500}\}$ and $|\mathbb{D}_S| = 50$.

4.2. 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 webpages, *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 $\mathcal{S} = \langle a_1, \ldots, a_i, \ldots, a_l \rangle$ as follows.

Definition 2 (*Launch Radius*). The launch radius $r_i(x)$ of an app x relative to a_i in $\mathcal{S}$ is defined as k iff

(i) $x = a_j$ in $\mathcal{S}$.
(ii) $k = |i - j|$.
(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 $\mathcal{S} = \langle a_1, \ldots, a_l \rangle$, 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.

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

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.

Definition 3 (*Clustering Affinity*). The clustering affinity $ca(x, y)$ between two apps x and y is defined as follows.

$$ca(x, y) = \frac{\overline{r(y|x)} \cdot |\mathcal{S}_x| + \overline{r(x|y)} \cdot |\mathcal{S}_y|}{|\mathcal{S}_x| + |\mathcal{S}_y|}.$$

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

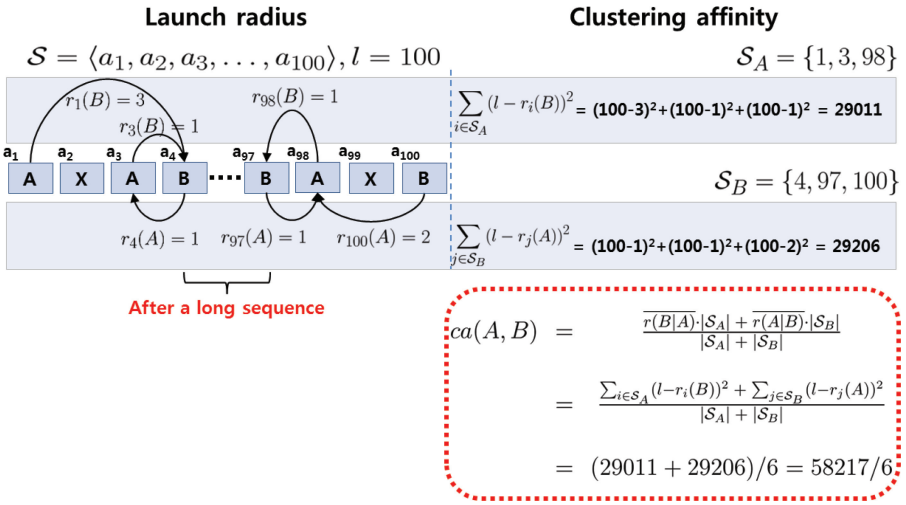


Fig. 7. An example of the launch radius and the clustering affinity between two apps.

For example, in Figure 7, 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 7, $ca(A, B)$ 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.

After all the cluster affinity values are computed over all (X, Y) pairs in $X, Y \in \mathbb{D}_{\mathcal{S}}$, the C-AUM heuristic clusters apps using the *single-linkage clustering* algorithm [Sneath 1957]. Figure 8 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* become *Cluster 4*.

Since clusters are identified in the order of decreasing cluster affinity values, earlier identified clusters in the preceding algorithm are regarded as more strongly related apps over later identified clusters. For example, in Figure 8, *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 app, we first introduce some related concepts needed in computing *cscore* values. Let a sequence $\mathcal{C} = \langle c_1, c_2, \ldots, c_z \rangle$ represent a cluster-construction sequence, where

[2]In this article, we do not include a detailed description of the *single-linkage clustering* algorithm, which can be found in Sneath [1957].

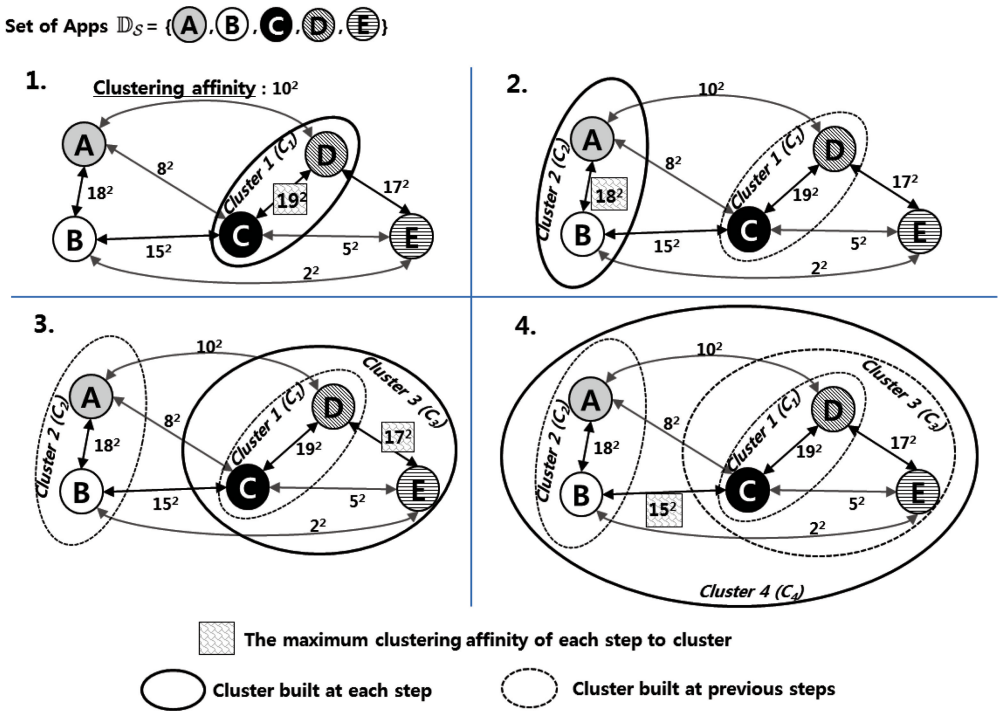


Fig. 8. An example of building clusters using the single-linkage clustering algorithm.

c_i indicates the ith 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 8, $\mathcal{C} = \langle c_1, c_2, c_3, c_4 \rangle$ 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 $\mathcal{C} = \langle c_1, \ldots, c_z \rangle$ as follows.

Definition 4 (*Start Cluster*). The start cluster $\alpha(x)$ of an app x is defined as c_k iff

(i) $x \in \mathbb{E}_{c_k}$.
(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 8, the start cluster of *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) $\mathbb{E}_c \subset \mathbb{E}_{c_k}$.
(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 8, 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 $\mathcal{C} = \langle c_1, \ldots, c_z \rangle$ is defined as c_k iff

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

ALGORITHM 1: 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
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
    end
    for each cluster c′ in ℂ do
        if c′ ≠ c_z then
            if ρ(c′) = ρ(c) then
                ℂ ← ℂ − {c′} ;
            end
        end
    end
    ℂ ← ℂ ∪ {ρ(c)} ;
    score ← score + 1 ;
end
```

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

Using these support functions, Algorithm 1 describes how the $cscore(x)$ of each app x is computed for a given cluster-construction sequence of clusters $\mathcal{C} = \langle c_1, \ldots, c_z \rangle$ and a given set $\mathbb{A} = \{a_1, \ldots, a_n\}$ of apps. (In order 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 $\mathcal{S} = \langle a_1, \ldots, a_l \rangle$, $\mathbb{A} = \{a_l, a_{l-1}, \ldots, a_{l-n+1}\}$.)

For example, shown as Step 4 in Figure 8, if *Apps* A and E are selected as $\mathbb{A} = \{A, E\}$, Algorithm 1 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 $\mathcal{S} = \langle a_1, a_2, \ldots, a_l \rangle$ can be computed as follows.

(i) Computing the cluster affinity values for all apps in $\mathbb{D}_{\mathcal{S}}$: $O(l^2)$.
(ii) Building clusters using the single-linkage clustering algorithm: $O(|\mathbb{D}_{\mathcal{S}}|^2)$, 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 $\mathbb{D}_{\mathcal{S}}$: $O(|\mathcal{C}| \cdot |\mathbb{D}_{\mathcal{S}}|)$.

Since the app sequence length l is much larger than $|\mathbb{D}_{\mathcal{S}}|$, 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 2,500 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., $|\mathbb{D}_{\mathcal{S}}| = 50$), 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 $O(|\mathbb{D}_{\mathcal{S}}|^2)$. In our current implementation on the Nexus S, it took on average 11.43ms to run the C-AUM heuristic every single time a new app was launched based on the real usage log, for the given $\mathcal{S} = \{a_1, \ldots, a_{2500}\}$ and $|\mathbb{D}_{\mathcal{S}}| = 50$ using $|\mathbb{A}| = 3$ in computing the *cscore* values of Algorithm 1.

5. APP LAUNCHING EXPERIENCE OPTIMIZATION

5.1. Motivation

As a specific example of personalized optimization, in the remainder of this article, we present an app launching experience optimization technique based on the app usage models. Since the most smartphone users expect their smartphones to be always on, always-connected devices, prompt response to user inputs to smartphones is an important design requirement that affects the quality of user experience. 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 article, we broadly define *app launching experience* as a type of user experience related to app launching in general.

For better app launching experience, 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, inactive apps are kept as background apps in the main memory of smartphones, thus, they can be quickly responded when the apps are launched again in the future. Background apps are evicted from the memory when the systems decide that they need more memory for, say, a new app. 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 relaunched 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.

5.2. Impact of Cold Starts on App-Launching Experience

5.2.1. 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 seven 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

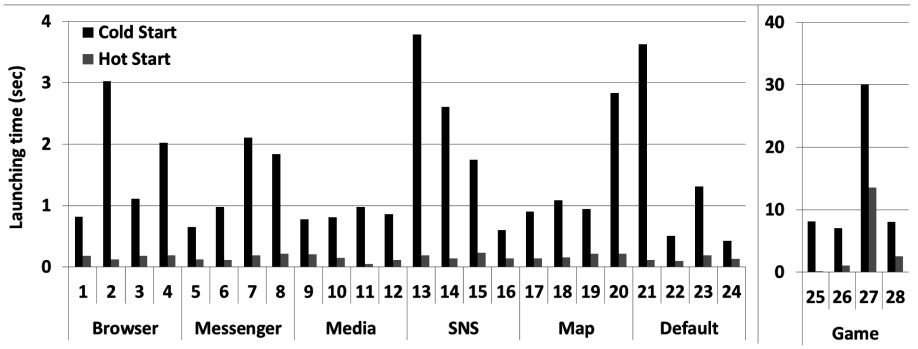


Fig. 9. Launching time differences between hot and cold starts.

precisely measure the launching completion time, because many apps start to react to 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 inhouse tool [Lim et al. 2011] was developed to measure launching times from this new definition.

Figure 9 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 9, the launching time of the cold start is on average nine 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 users uncomfortable [Tolia et al. 2006].) 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.

5.2.2. Energy Consumption Difference between Hot and Cold Starts. As previously mentioned, 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 Zhang et al. [2010].

Figure 10 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 10 shows differences in power consumption. In Figure 10, the X-axis and the Y-axis represent time in millisecond and the electrical current, respectively. As shown in Figure 10, the extra energy consumption caused by the cold start is observed significantly higher than that of the hot start.

5.2.3. 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

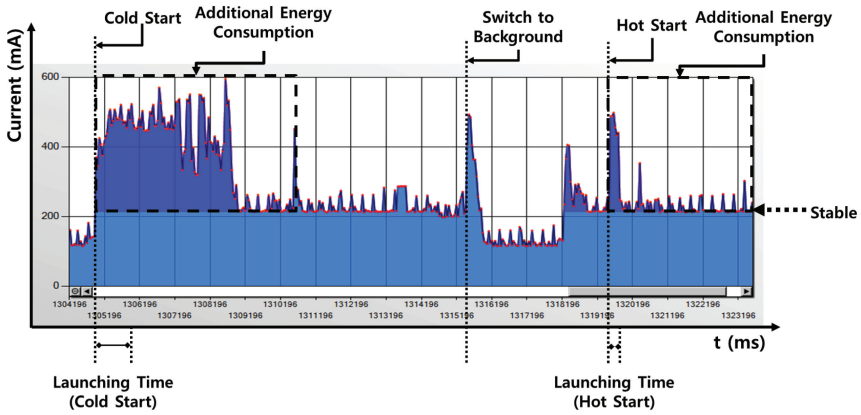


Fig. 10. Current changes during the launching process.

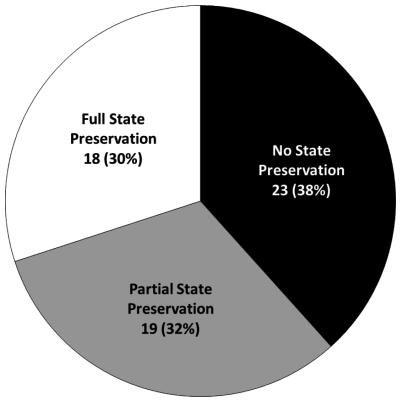


Fig. 11. A breakdown of 60 apps based on the degree of state preservation.

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 11, 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.

6. AUM-BASED USER EXPERIENCE OPTIMIZATION

6.1. 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 occured 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

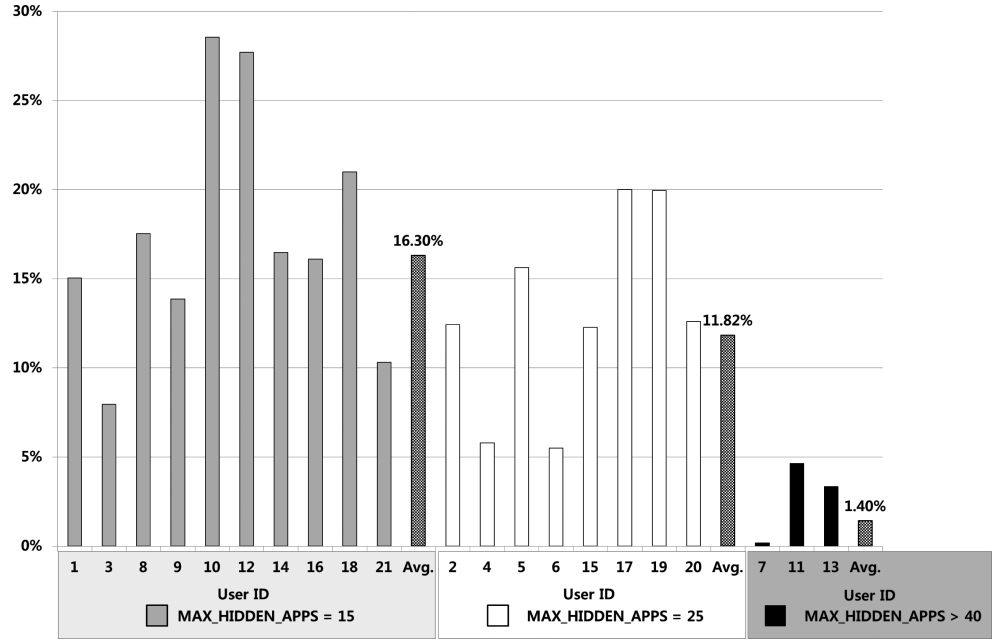


Fig. 12. Restart ratio distributions of 21 users.

platform. It limits the number of the background processes (which are called hidden apps[3]) 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.

6.2. Problem of the LRU-Based Task Killer

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

Figure 12 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 (*user 3*) has experienced the restart ratio less than 10%. Figure 12 also shows the effect of the MAX_HIDDEN_APPS value on the restart ratio. In the 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, a large number of background processes will incur other overhead in memory management with a risk of continuous memory leaks from

[3]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.

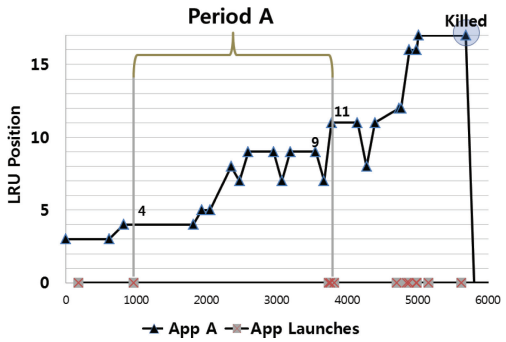


Fig. 13. Changes in LRU stack positions of *App A* over time.

poorly-behaving tasks. Therefore, it is important to minimize the restart ratio under a small MAX_HIDDEN_APPS value.

6.2.2. 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 5.2. 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.

6.2.3. 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 13 shows how the LRU stack position of *App A* varies until *App A* is killed by the LRU-based task killer. Although none of the other apps were launched during `Period A`, the LRU stack position of *App A* switches from position 4 to 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.

6.3. AUM-Based Optimization 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 previously mentioned, 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 previously), we implemented the AU-aware task killer which selects a victim based on our AUM.

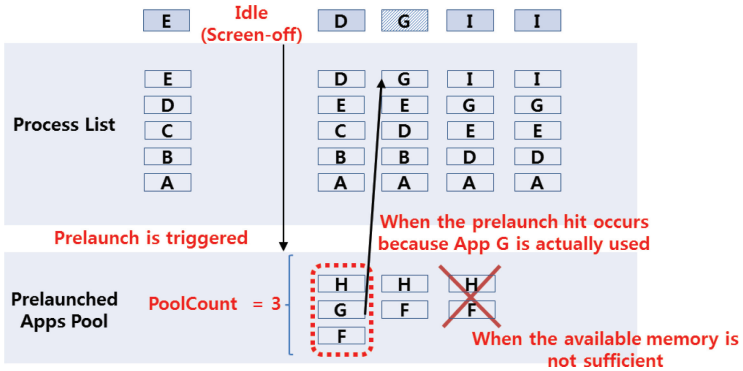


Fig. 14. The overview of the AU-aware prelauncher.

Algorithm 2 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 nondecreasing 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.

ALGORITHM 2: 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
AppListSortedByScore ← stableSortByScore(ScoredAppList);
while (AppListSortedByScore.size() > MAX_HIDDEN_APPS) do
    victimApp ← getHighestScoreApp(AppListSortedByScore);
    killProcess(victimApp);
end
```

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

6.4. AUM-Based Optimization 2: AU-Aware Prelauncher

As another approach to improving 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 14. The number of currently prelaunched apps, called as *PoolCount*, is decided depending on the amount of available

Table I. 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

memory, as in Equation (1).

$$PoolCount = \frac{AvailMem\text{-}SpareMem}{AvgMemOfApp}. \quad (1)$$

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 14, 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.

7. EXPERIMENTAL RESULTS

7.1. 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 section.) The prelaunching technique, which was described in Section 6.4, 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 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 the 21 user logs we have collected, we have selected four usage logs as they represent typical usage scenarios in terms of the restart 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 I summarizes the main characteristics of these 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.

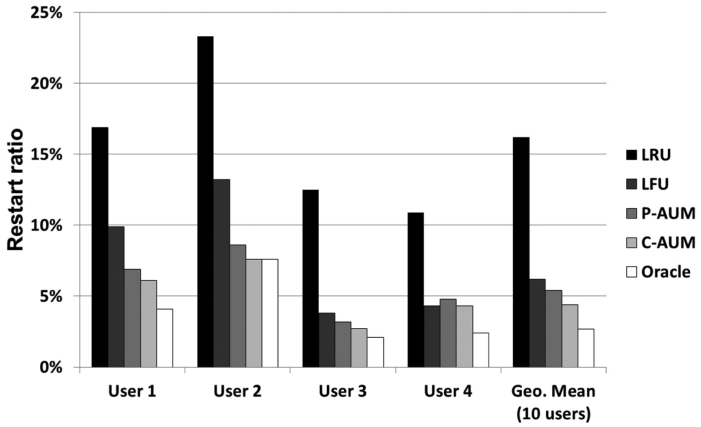


Fig. 15. Restart ratio comparisons of five policies.

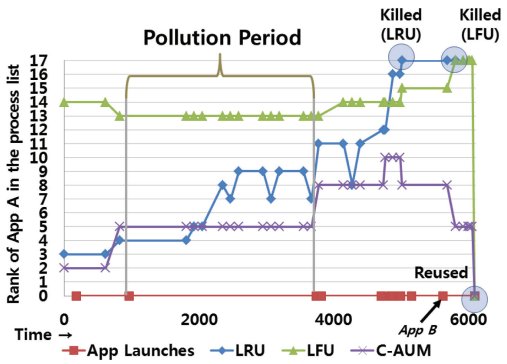


Fig. 16. Changes in the rank of *App A* over time.

7.2. Results of Task Killing Mechanism Optimization

7.2.1. Restart Ratio Comparisons. Figure 15 compares the restart ratios of five different task killing policies for four representative users. As shown in Figure 15, 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 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, the more important the app. As shown in Figure 16, the LRU policy is vulnerable to the LRU stack pollution problem (as discussed in Section 6.2.3). 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 reused. In the LFU-based policy, since *App A* was not frequently 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

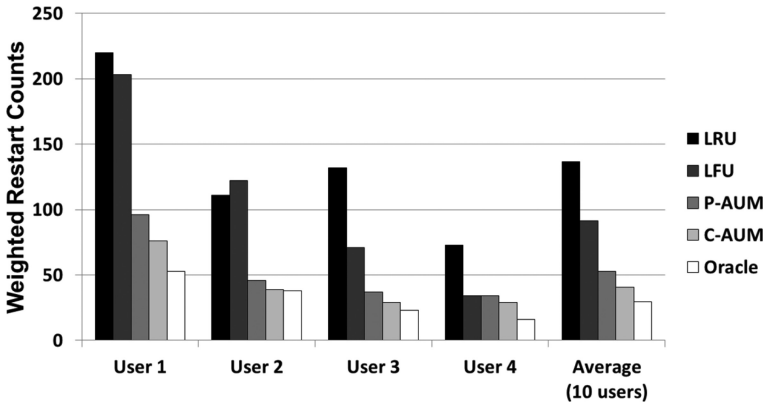


Fig. 17. Weighted restart count comparisons of five policies.

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 between two consecutive screen-off points.) Figure 17 compares weighted restart counts for five policies. Compared to the restart ratio comparison result of Figure 15, 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 can. 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.

7.2.2. Impact on App Launching Experience. As mentioned in Section 5.2, when a cold start of an app occurs, it can adversely affect app launching experience over a hot start from the 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 18 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 19 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 energy consumption of 19.79% and 22.48%, respectively.

Figure 20 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.

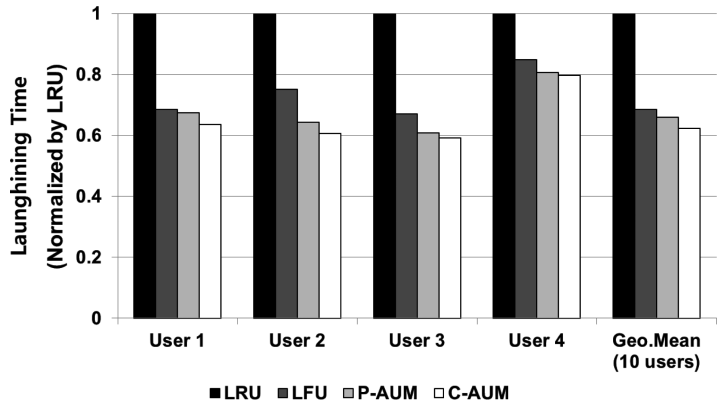


Fig. 18. Normalized launching time comparisons of four policies.

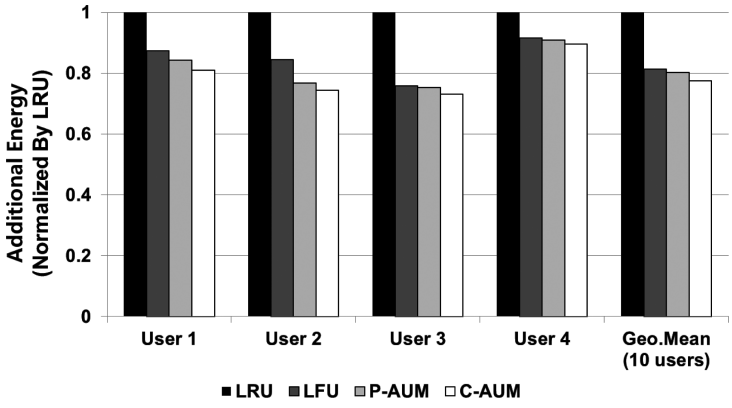


Fig. 19. Normalized energy consumption comparisons of four policies.

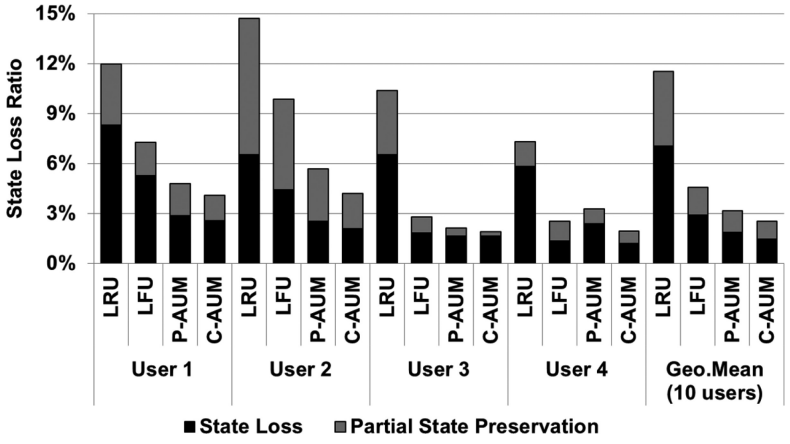


Fig. 20. State loss ratio comparisons of four policies.

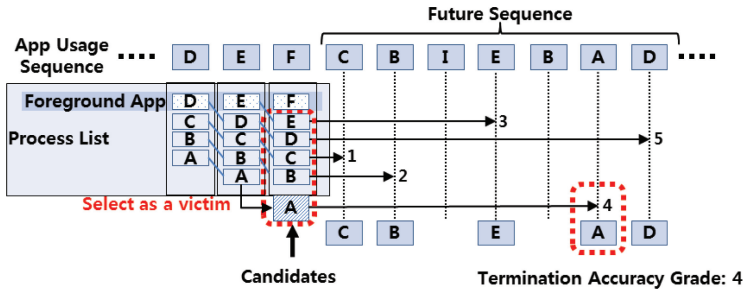


Fig. 21. An example of computing termination accuracy grades.

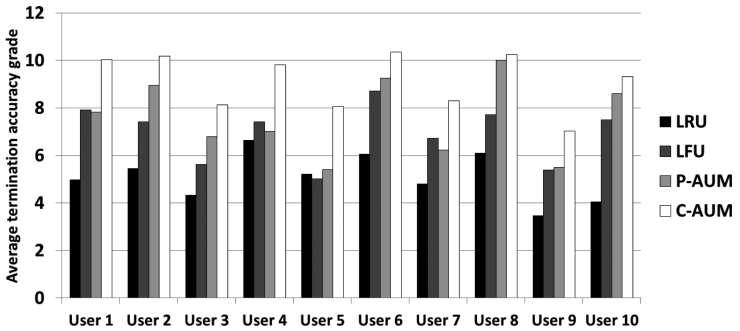


Fig. 22. Comparisons of the average termination accuracy grade of four policies.

7.2.3. Evaluation of Termination Decision Accuracy. As mentioned in Section 6.1, 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 21, which are currently in the process list. Since a task killing policy operates under the assumption that the selected victim will not be reused for a long time; counting the number of apps launched before the victim app is relaunched in the future can give 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 21 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 relaunched in the order of *App C*, *B*, *E*, *A*, and *D*.

In order to evaluating the accuracy of termination decisions of different policies, termination accuracy grades are calculated every termination decision time for the different task killing policies, including the LRU-based, LFU-based, P-AUM-based, and C-AUM-based policies. As shown in Figure 22, 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.

7.3. Results of Prelaunching Technique

Figure 23 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 23, reduces the restart ratio by up to 21.3% over the C-AUM-based policy.

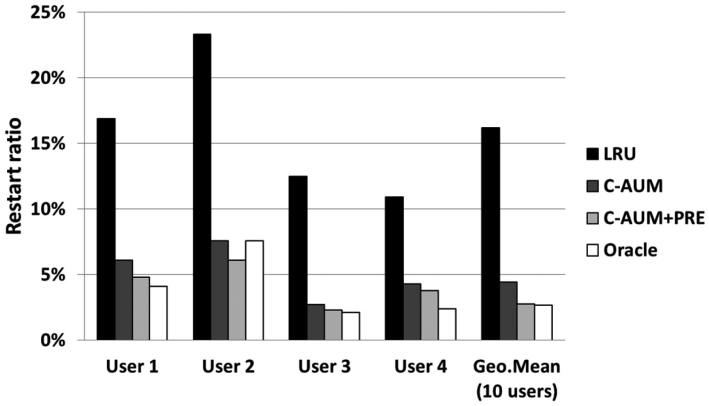


Fig. 23. The effect of the prelaunching technique on the restart ratio.

It is interesting to note that the C-AUM+PRE policy even outperforms the oracle policy (the Oracle policy in Figure 23) that makes task termination with 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 is enough memory available to do so. On the other hand, the Oracle policy is limited to MAX_HIDDEN_APPS apps.

8. RELATED WORK

8.1. 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 [Shye et al. 2010; Falaki et al. 2010; Kiukkonen et al. 2010; Do et al. 2011]. For example, Shye et al. [2010] observed that only a few states and transitions are required to build a user activity model on the smartphone usage behavior. Falaki et al. [2010] 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 (described in Section 2) also agrees with the findings of Shye et al. [2010] and Falaki et al. [2010]. 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.

8.2. Launching Time Optimization

Many research groups have investigated reducing the application launching time because it was considered as one of the important user-perceived performance metrics. 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 [Microsoft 2007; Esfahbod 2006] for HDDs cannot be directly applied to smartphones.

Recent investigations have more directly focused on improving application launch performance on NAND flash-based storages, such as solid state drives (SSDs) [Ryu et al. 2011; Baiocchi and Childers 2011]. For example, Joo et al. presented an SSD-aware application prefetching scheme, called FAST [2011]. FAST exploits the fact that the I/O time can be overlapped with the computation time during the application launch procedure. While existing techniques are effective in reducing the launching time by intelligently exploiting the underlying devices' characteristics, our approach is fundamentally different in that we take advantage of high-level information (such as app usage patterns) in optimizing the launching time.

9. CONCLUSIONS

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 runtime and enables more advanced and effective optimizations for smartphones. In this article, we have developed a couple of app usage models which can be used in predicting a 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.

Our work can be extended in several directions. For example, we can extend our proposed AUMs to include different types of context information (e.g., location and time). With an extended AUM, we can make more intelligent context-aware decisions in managing system resource. Our current app launching optimization technique can be also extended, for example, to distinguish apps with long restart times from ones with short restart times. For apps with very short restart times, it may be better to terminate them instead of keeping them in memory as background apps.

REFERENCES

J. A. Baiocchi and B. R. Childers. 2011. Demand code paging for NAND flash in MMU-less embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*.

Digitizor. 2011. Android stats: 200k market apps, 400k new activations daily, malware up by 400%. http://digitizor.com/2011/05/11/android-stats/.

T. M. T. Do, Jan Blom, and Daniel Gatica-Perez. 2011. Smartphone usage in the wild: A large-scale analysis of applications and context. In *Proceedings of the International Conference on Multimodal Interaction*.

B. Esfahbod. 2006. Preload - an adaptive prefetching daemon. Master's thesis. University of Toronto.

Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. 2010. Diversity in smartphone usage. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*.

Google. 2010. Nexus s. http://www.google.com/phone/detail/nexus-s.

Y. Joo, J. Ryu, S. Park, and K. G. Shin. 2011. Fast: Quick application launch on solid-state drives. In *Proceedings of the USENIX Conference on File and Stroage Technologies*.

N. Kiukkonen, J. Blom, O. Dousse, Daniel Gatica-Perez, and Juha Laurila. 2010. Towards rich mobile phone datasets: Lausanne data collection campaign. In *Proceedings of the ACM International Conference on Pervasive Services*.

V. I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10, 8, 707–710.

Jehun Lim, Hakbong Kim, Wook Song, and Jihong Kim. 2011. Ltmeter: An app launching time analyzer for personal smart devices. In *Proceedings of the International Conference on Ubiquitous Information Technologies & Applications*.

Microsoft. 2007. Inside the Windows Vista Kernel. http://www.microsoft.com/technet/technetmag/issues/2007/03/VistaKernel/.

J. Ryu, Y. Joo, S. Park, H. Shin, and K. G. Shin. 2011. Exploiting SSD parallelism to accelerate application launch on SSDs. *Electron. Lett.* 47, 5, 313–315.

A. Shye, B. Scholbrock, G. Memik, and P. A. Dinda. 2010. Characterizing and modeling user activity on smartphones: Summary. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.

P. H. A. Sneath. 1957. The application of computers to taxonomy. *J. Gen. Microbiol.* 17, 1, 201–226.

N. Tolia, D. G. Andersen, and M. Satyanarayanan. 2006. Quantifying interactive user experience on thin clients. *Computer* 39, 3, 46–52.

Wireless Intelligence. 2011. 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/.

Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. 2010. 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*.

Received January 2012; revised September 2012; accepted November 2012