CAS: Context-aware Background Application Scheduling in Interactive Mobile Systems

Joohyun Lee†, Member, IEEE, Kyunghan Lee‡, Member, IEEE, Euijin Jeong†, Student Member, IEEE, Jaemin Jo†, Student Member, IEEE, and Ness B. Shroff‡, Fellow, IEEE

Abstract—Each individual’s usage behavior on mobile devices depend on a variety of factors such as time, location, and previous actions. Hence, context-awareness provides great opportunities to make the networking and the computing capabilities of mobile systems to be more personalized and more efficient in managing their resources. To this end, we first reveal new findings from our own Android user experiment: (i) the launching probabilities of applications follow Zipf’s law, and (ii) inter-running and running times of applications conform to log-normal distributions. We also find contextual dependencies between application usage patterns, for which we classify contexts autonomously with unsupervised learning methods. Using the knowledge acquired, we develop a context-aware application scheduling framework, CAS that adaptively unloads and preloads background applications for a joint optimization in which the energy saving is maximized and the user discomfort from the scheduling is minimized. Our trace-driven simulations with 96 user traces demonstrate that the context-aware design of CAS enables it to outperform existing process scheduling algorithms. Our implementation of CAS over Android platforms and its end-to-end evaluations verify that its human involved design indeed provides substantial user-experience gains in both energy and application launching latency.

Index Terms—Context-awareness; Context-aware networking and computing; Application unloading/preloading; Start-up latency; Energy minimization

I. INTRODUCTION

As mobile devices have become an essential part of our lives, people expect more capability from them such as longer battery life, ubiquitous access to Internet, immediate response time, and fresh contents (e.g., messages, feeds, news, ads, sync data, or software updates). The recent advancement of cellular networks and cloud computing is partly fulfilling these needs. However, certain performance features such as long battery life and high quality-of-service (e.g., low latency and freshness) have intrinsic tradeoffs that make it difficult to optimize simultaneously.

In a large-scale measurement study of 2000 Galaxy S3 and S4 devices by Chen et al. [2], [3], 45.9% of the total energy drain occurs during screen off periods. This high energy consumption mainly comes from background applications that update contents, collect user activity information, or keep components in active states [4], [5]. However, these background activities may not be always beneficial for users. For example, if a social network application updates its contents frequently (say every 20 minutes), but the user launches this application once a day, then most updates unnecessarily waste network energy.1 As a motivational example, we show the measured daily network usage2 of a Facebook application on a Galaxy S7 smartphone running Android 6.0.1 in Fig. 1, where the update or collection intervals are less than 20

Fig. 1. Daily network usage of the Facebook app and its corresponding state either being in the foreground or background. The Facebook app incurs background network traffic even when the user is not interacting with it.

Fig. 2. Measured power consumption of a popular game application for foreground and background states in a Galaxy Note 2 smartphone.

1It is well known that frequent network traffic incorporates large ramp and tail energy overheads [6], [7].
2We log network usage by reading /proc/uid_stat/[uid]/.

1J. Lee is with the Department of Electrical and Computer Engineering at The Ohio State University; e-mail: lee.7119@osu.edu.
2K. Lee, the corresponding author, E. Jeong, and J. Jo are with the Department of Electrical and Computer Engineering, UNIST (Ulsan National Institute of Science and Technology), South Korea, e-mails: {khlee, ujeong, jaemin}@unist.ac.kr.
3N. B. Shroff holds a joint appointment in both the Department of Electrical and Computer Engineering at The Ohio State University; e-mail: shroff.11@osu.edu.
4This research was supported in part by grants from the National Science Foundation CNS-1409336 and CNS-1446582, and the Army Research Lab MURI W911NF-12-1-0385. This research was also supported in part by NRF grant (NRF-2016R1D1A1B03935042) funded by the Korea government (MSIP) and the research funds of UNIST (grant 1.160048.01 and grant 1.1704.01).
5A preliminary version of this work was presented at ACM UbiComp 2016 [1].
minutes. Also, gaming or map applications often keep high power-consuming components such as CPU and GPS in active states while being in background. This operation is intended to provide immediate responses from those applications but wastes energy unless the user re-launches them within a short time. This inefficient stand-by operation is indeed observed in a popular game application, as shown in Fig. 2. To this end, we aim at managing mobile applications in a resource-efficient manner by exploiting per-user application usage behaviors analyzed in the perspective of contextual usage statistics. It is important to mention that managing applications not only influences the computing behaviors but also the networking behaviors of a mobile system which in turn leads to further resource optimizations such as delaying or suppressing non-urgent background network traffic.

To our knowledge, the most widely used application controller in Android [8] and iOS [9] is called the low memory killer (LMK) that commonly kills (i.e., unloads or terminates) applications to secure more available memory. Popular memory kill algorithms that are often implemented with LMK purge applications in the order of either LRU (least recently used) or process priority [10]. As this mechanism is merely inherited from computer systems with abundant resources (e.g., energy), it never considers contextual information of application usages. Thus, it naturally fails to manage mobile applications in an efficient way.

There have been two complementary approaches to tackle this problem. Several papers [11]–[14] tried to identify energy bugs/hogs, that mainly come from coding errors. This may successfully kill all detected buggy activities, but benign operations such as activity logging can also be stopped (false positive) and unnecessary network activities may be mostly intact (false negative). Another recent approach in [2] proposed a metric called BFC (Background to Foreground Correlation) to quantify the level of user engagement for each application on the fly. If the BFC value is smaller than a threshold, background activities are implemented to be suppressed. [2] also developed HUSH that puts applications that have not been recently used in foreground into inactive states, and extends the duration of being in the inactive states in an exponential manner. They showed that the screen-off energy saving of their algorithms is 15-17% in their large-scale traces.

The second approach partly tackled the energy-inefficient activities, but still this approach is myopic as it ignores the very important statistics on when the user will relaunch an application. As human behaviors have regular patterns in their daily lives, it is clearly possible to design a more efficient application controller that is far beyond the naive exponential mechanism. This is only possible when deeper understandings of per-user and per-application usage behaviors are acquired.

To that end, we collect application usage of 103 Android users for which we deployed a logger that was designed to periodically send detailed application, sensor, and memory usage data to our server. The total data collected spans over 1057 days and reaches about 20GB. We find that the usage patterns follow heavy tail distributions: (i) The launching probabilities of applications follow the Zipf’s law, and (ii) inter-running and running times of applications resemble log-normal distributions. We also reveal detailed context-dependency in the re-launching probabilities, which convey more personalized control ideas over existing studies [15]–[19]. To realize a control algorithm that exploits such personalized context-dependency, we automate the procedure of per-user context extraction by adopting unsupervised learning methods that significantly improve prediction accuracy.

With the contextual knowledge, we propose a new application control framework, CAS (Context-aware Application Scheduler) that works by predicting when a user will launch an application and which application will be used. Trace-driven simulations with consideration of system overhead show that CAS outperforms the Android genuine resource scheduler, LMK, and Android 6.0. We also verify the practicality of CAS by implementing the system on Android.

II. RELATED WORK

We classify previous work on mobile resource scheduling into several categories from experimental studies to implementations and summarize their contributions.

Human behaviors on mobile application usage: To establish the foundation of context-awareness for mobile resource scheduling, several pioneering experimental studies [15]–[21] have been performed to analytically understand how humans use applications given contexts such as time/location information, and the last used application. Falaki et al. [21] studied usage traces from 255 users and found that the levels of activities are vastly different across users. They also found that screen off times fit well with the Weibull distribution.

Application preloading algorithms: Those early studies on context-awareness led to the development of application preloading/prefetching algorithms [22]–[26] applications that substantially reduce the perceivable start-up latency (i.e., launch latency) by preparing required resources (including computation such as rendering, and communication such as feed updates) before they are requested by users. However, most previous studies have focused on which application a user will launch next, but not on when the user will launch it. [23] is the only work that concerned the moment of launching, but the authors did not consider the cumulative penalty of preloaded applications, hence their prefetching schedules may suffer from large energy wastage until the predicted application is actually accessed.

Application unloading algorithms: The default low memory killers (LMK) on Android [8] and iOS [9] unload or terminate applications to secure more memory resource, when the available memory goes below a pre-defined threshold.
Popular memory kill algorithms that are often implemented with LMK purge applications in the order of either LRU (least recently used) or process priority [10]. Android version 6.0 (Marshmallow), released in October 2015, adopts new features called App standby and Doze mode [27] for energy saving. App standby suppresses background activities of an application that has not been used in foreground for 3 days. The Doze mode is enabled when a user leaves the device for a certain amount of time. Doze mode restricts background apps’ access to network and CPU for most of the time, and lets background apps complete their activities for a short maintenance window. Doze mode schedules this maintenance window less frequently as the untouched period gets elongated.

A recent paper [2] proposed simple unloading algorithms called BFC (Background to Foreground Correlation) and HUSH for screen-off background activities. The BFC metric quantifies the likelihood that a user will interact with an application during a next screen-on interval after its background activities. BFC updates the metrics using an exponential moving average at the end of each screen on period, and unloads applications if their BFC metrics are less than a cutoff value $\alpha$. Another algorithm, HUSH increases the suppression interval of an application if it has not been used in foreground using exponential backoff (i.e., the interval is multiplied by a given scaling factor $\sigma$). Once an application is used in foreground, the interval is reset to an initial value. This simplistic algorithm is shown to save about 15-17% of energy in their large-scale usage traces. Our preliminary work [1] was the first of its kind that jointly considers preloading and unloading of background applications. However, the scheduling algorithm therein was not able to systematically find an optimal schedule for a given resource constraint (e.g., energy, or launching latency).

III. PRELIMINARY

In this section, we explain basic concepts for application processes. In Android OS, there are various application states each of which has its corresponding “process importances” ranging from 100 to 1000 [28]. An Android application installed on a device stays in one of the states at a time slot. Fig. 3 shows all the states defined in Android and our simplified mapping of those states into three states: foreground, background, and empty. We define a foreground process to be a process in use and that is visible to users. By the definition, there can be at most one application in foreground at each time. An empty process is defined to be a process unloaded from memory, and thus no resource is allocated to that process. We denote a background process as a process that is loaded but not running on foreground.

The rationale behind our simplification of states is that the processes that are running with no foreground UI on the screen show similar resource consumption characteristics (e.g., memory and power) as background processes of importance 400 rather than foreground processes running on the screen. Also, these processes can be unloaded just like background processes of importance 400 without disrupting on-going user experience, except system processes (e.g., phone caller and application launcher) that are designed to be running all the time, and user-interactive applications (e.g., music, radio and recorder) that are usable even without visible UIs.

We depict the transitions between states in Fig. 4. An empty to foreground transition called cold launch occurs when a user touches an empty (i.e., unloaded) application to launch. A transition from background to foreground called warm launch is mostly made when a user chooses to use the application by re-launching an application that is still kept in the background, and thus has shorter latency than cold launch but consumes memory and battery for background activities. Therefore, user experience on battery life and application launch latency is highly dependent on the decision of putting an application in either of background or empty state.

We further define the system state as either of off or on and its period. $T_k^{off}$ denotes the $k$-th screen-off period when all applications are either in background or empty, while $T_k^{on}$ denotes the $k$-th screen-on period for which an application is being used in the foreground. Fig. 5 depicts how the number of background applications ($|B(t)|$) changes as the screen state and foreground application $X_k$ change over time, under the Android default scheduler LMK, where $B(t)$ and $X_k$ denote the list of background applications at time $t$ and the foreground application at $k$-th screen on period. Under LMK, a foreground process goes to background when the user switches

\[\text{foreground, background, empty. We define a foreground process to be a process in use and that is visible to users. By the definition, there can be at most one application in foreground at each time. An empty process is defined to be a process unloaded from memory, and thus no resource is allocated to that process. We denote a background process as a process that is loaded but not running on foreground.}\]

\[\text{The rationale behind our simplification of states is that the processes that are running with no foreground UI on the screen show similar resource consumption characteristics (e.g., memory and power) as background processes of importance 400 rather than foreground processes running on the screen. Also, these processes can be unloaded just like background processes of importance 400 without disrupting on-going user experience, except system processes (e.g., phone caller and application launcher) that are designed to be running all the time, and user-interactive applications (e.g., music, radio and recorder) that are usable even without visible UIs.}\]

\[\text{We depict the transitions between states in Fig. 4. An empty to foreground transition called cold launch occurs when a user touches an empty (i.e., unloaded) application to launch. A transition from background to foreground called warm launch is mostly made when a user chooses to use the application by re-launching an application that is still kept in the background, and thus has shorter latency than cold launch but consumes memory and battery for background activities. Therefore, user experience on battery life and application launch latency is highly dependent on the decision of putting an application in either of background or empty state.}\]

\[\text{We further define the system state as either of off or on and its period. $T_k^{off}$ denotes the $k$-th screen-off period when all applications are either in background or empty, while $T_k^{on}$ denotes the $k$-th screen-on period for which an application is being used in the foreground. Fig. 5 depicts how the number of background applications ($|B(t)|$) changes as the screen state and foreground application $X_k$ change over time, under the Android default scheduler LMK, where $B(t)$ and $X_k$ denote the list of background applications at time $t$ and the foreground application at $k$-th screen on period. Under LMK, a foreground process goes to background when the user switches}\]
to a different foreground application or turns off the screen. LMK kills applications in background in the descending order of importance values when the available memory goes below multiple levels of preset memory thresholds. This is surely done with no consideration on when the killed application is going to be relaunched. Thus, LMK results in higher cold launch probability, even though it keeps a number of applications in background and brings high energy wastes.

IV. MEASUREMENT STUDY

A. Data collection

To capture application usage behaviors of smartphones in the wild, we performed our own data collection with 96 Android users selected from a few popular Internet communities of South Korea during two weeks in Feb. 5-18, 2015. We provided a data logger programmed to periodically record application usage and device characteristics summarized in Table I, and upload the data to our server daily. We anonymized all user information and IDs at the level of user devices. We asked users not to use task killers and not to manually unload applications while participating our experiment, in order to see how the Android genuine scheduler, LMK works. The average valid data per user is about 11 days, and the total data size is about 20GB. We also asked the participants to fill an anonymized survey involving occupation, age band, gender, and personal statement on their dissatisfaction of the smartphone (e.g., latency, freeze), summarized in our survey report [29]. To improve the reliability of the responses we did our best to create an anonymous interface to give them confidence in providing the correct information. Participants come from diverse occupations, genders, ages, and devices (e.g., Samsung Note2, Note3, Note4, S3, S4, S5, LG G2, G3). Most of participants use Android KitKat (4.4.2) (75%), where a small number of them use Jelly Bean (4.2.2 and 4.3) and Lollipop (5.0.1). From our survey, short lifetime, frequent freezes, and long start-up latency were still the major problems for participants, even though their smartphones were mostly state-of-the-art.

B. Key observations from the measurements

We summarize key observations in this subsection.

Application usage statistics of users and states: In Fig. 6, we plot the fraction of time spent in different process importance evaluated from our experimental logs, the number of running processes at a moment, and the number of unique processes that have ever been used during the experiment. We treat system and user-interactive (e.g., music) processes separately in the figure. We find that the number of running (foreground+background) processes per user is 5.2 on average, and the number of unique processes ever used per user is 55.1 on average, excluding system and user-interactive processes. 73% of unique processes have not been used in foreground for more than 3 days in our traces, and these processes will be unloaded by the new feature App standby6 of Android 6.0 released in late 2015, which suppresses background activities of an application that has not been used in foreground for 3 days. However, the number of corresponding background processes in run is only 2 on average (40% of that in LMK) so that the energy saving from this feature is not significant as we will see in our simulation section. The fraction of time a process spends in the foreground state is about 6% on average, while the fraction of time in background is about 16 times of being in foreground. The fraction of time that the screen is on is 21% on average (i.e., 5 hours per day).

Regularity in application usage: The existence of the regularity of application usage patterns of a person is the key to make a mobile system predictive, and thus more efficient. In order to understand individual application usage patterns, we investigate the timings of all foreground application actions (launching/stopping) and analyze the event intervals. For the visualization, we choose one user randomly and depict the timings of the application launches for two consecutive weeks in Fig. 7. We observe that the active hours are highly regular and the intensity of activities during the weekdays or weekends for two weeks resemble each other. More specifically, we find that there exists strong distributional similarity in both off and on periods in the first week and the second week, as shown in Fig. 8. These results confirm that temporal and distributional knowledge from usage history can be used to better predict the future application usage.

6Our measurement is conducted before this feature is provided.
The probability density function (PDF) of the log-normal distribution with parameters $\mu$ and $\sigma$ is 
\[ f(x; \mu, \sigma) = \frac{1}{x \sigma \sqrt{2\pi}} \exp\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right). \]

The cumulative distribution function (CDF) of $T$, respectively.

Off/on period distribution: In Fig. 9, we fit off/on period distributions of a randomly chosen user to show that the distributions are heavy-tailed. We verify by Cramer-Smirnov-Von-Mises (CSVM) [30] and Akaike [31] tests that off/on periods of all users have the best fit with log-normal distributions\(^7\) rather than exponential, Weibull, truncated Pareto, gamma and Rayleigh distributions. We use the best fitting log-normal distributions as representative of off/on periods in the following sections for tractability. We also depict the CDFs of average individual off/on period of users in Fig. 10. The average individual off period in total is 15.5 mins for a whole day, 13.5 mins for the active hours (9:00 to 24:00) and 33.8 mins for the inactive hours (24:00 to 9:00). Not surprisingly, the off period in the inactive hours is much longer than in the active hours, as users tend to leave the device unattended during the inactive hours. The average individual on period is about 1.4 mins.

Off/on failure rates: In order to deeply understand the application usage behavior, we quantify the frequency of altering its state from “off to on” (launching) or from “on to off” during off/on periods at the elapsed time $t$, which is commonly called as the failure rate. Formally, the failure rate of $T$ is 
\[ \hat{r}_T(t) \triangleq \frac{f_T(t)}{F_T(t)}, \] for $t$ such that $F_T(t) < 1$, where $f_T(t)$ and $F_T(t) = \mathbb{P}[T \leq t]$ are the probability mass function and cumulative distribution function (CDF) of $T$, respectively. $T$ can be either $T^{\text{off}}$ or $T^{\text{on}}$. We call off failure rate (from off to on) for $T^{\text{off}}$ and on failure rate (from on to off) for $T^{\text{on}}$. In Fig. 11, we plot off and on failure rates, for each user (dotted lines) and on average (solid line). For most of users, the off and on failure rates increase at first but soon decrease right after 10 seconds. The pattern of having decreasing failure rate over time is called negative aging [32]. This indicates that
TABLE II
THE PORTION OF USER-TRIGGERED LAUNCHES, AVERAGE RUNNING TIMES OF 12 MOST POPULAR APPLICATIONS AND THEIR top-1 TO top-3 PROBABILITIES ACROSS ALL USERS.

<table>
<thead>
<tr>
<th>category</th>
<th>process name</th>
<th>launches</th>
<th>time</th>
<th>top-1</th>
<th>top-2</th>
<th>top-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messaging</td>
<td>com.kakao.talk</td>
<td>27%</td>
<td>47s</td>
<td>44%</td>
<td>25%</td>
<td>7.3%</td>
</tr>
<tr>
<td>Browsing1</td>
<td>com.android.browser</td>
<td>6%</td>
<td>161s</td>
<td>9.4%</td>
<td>9.4%</td>
<td>10%</td>
</tr>
<tr>
<td>Portal</td>
<td>com.nhn.android.search</td>
<td>4.4%</td>
<td>123s</td>
<td>2.1%</td>
<td>5.2%</td>
<td>9.4%</td>
</tr>
<tr>
<td>Browsing2</td>
<td>com.sec.android.app.shower</td>
<td>3.8%</td>
<td>126s</td>
<td>5.2%</td>
<td>5.2%</td>
<td>5.2%</td>
</tr>
<tr>
<td>Social</td>
<td>com.facebook.kakaua</td>
<td>3.3%</td>
<td>126s</td>
<td>-</td>
<td>6.3%</td>
<td>6.3%</td>
</tr>
<tr>
<td>Contacts</td>
<td>com.android.contacts</td>
<td>2.7%</td>
<td>26s</td>
<td>4.2%</td>
<td>1%</td>
<td>5.2%</td>
</tr>
<tr>
<td>Social</td>
<td>com.nhn.android.band</td>
<td>2.2%</td>
<td>49s</td>
<td>-</td>
<td>5.2%</td>
<td>1%</td>
</tr>
<tr>
<td>Browsing</td>
<td>com.android.chrome</td>
<td>2.2%</td>
<td>119s</td>
<td>3.1%</td>
<td>4.2%</td>
<td>4.2%</td>
</tr>
<tr>
<td>Setting2</td>
<td>com.android.settings</td>
<td>1.8%</td>
<td>29s</td>
<td>-</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>Social</td>
<td>com.nhn.android.naver.desenal</td>
<td>1.5%</td>
<td>82s</td>
<td>-</td>
<td>1%</td>
<td>2.1%</td>
</tr>
<tr>
<td>Game</td>
<td>com.supercell Clashofclans</td>
<td>1.5%</td>
<td>251s</td>
<td>-</td>
<td>2.1%</td>
<td>2.1%</td>
</tr>
<tr>
<td>Messaging</td>
<td>g.p.naver.line.android</td>
<td>1.2%</td>
<td>29s</td>
<td>2.1%</td>
<td>-</td>
<td>2.1%</td>
</tr>
</tbody>
</table>

1 Android default applications.
2 Messaging: 31.2%, Browsing: 14.5%, Portal: 11.7%, Social: 8.5%, Game: 4.9% (for 100 most popular applications).

Fig. 12. The CDF of the launching probability of $m$ most frequently used applications (left) and Zipf distribution fitting for the average launching probability (right). The frequently used applications of each user are not identical. The dotted lines are for each individual user.

users are less likely to launch an app as the off or on period increases. Thus, an energy-efficient control needs to reduce background activities as the failure rate starts to get reduced. This also suggests that the increasing backoff mechanisms of HUSH [2] and Doze [27] can be effective although their schedules are neither optimized nor personalized given that the individual failure rates (dotted lines in Fig. 11) show distinct characteristics for different users.

Frequently used applications: In Table II, we summarize the 12 most popular applications across all participants from the perspective of the launching probabilities, average running times and top-1 to top-3 probabilities. Top-$n$ probability of an application is defined as the probability that the application is the $n$-th most frequently used application of a user. The most popular application in our experiment is shown to be KakaoTalk (com.kakao.talk), a messaging application known as used by 93% of smartphone users in South Korea as of May 2014. 95% of our participants use KakaoTalk.

In Fig. 12, we depict the launching probability of frequently used applications of users. Note that the applications are individually sorted. We find that the launching probability follows Zipf’s law$^8$ with exponent $s = 1.4$, and the aggregated launching probability of the 10 most frequently used applications of a user is more than 80% on average. Recall that the average number of unique applications ever used for a user is 55.1. Therefore, users tend to use a small fraction of the applications most of the time, and there is little gain in the start-up latency and related user experience when infrequently used applications are kept in background.

Memory consumption: The average physical memory size of experimented smartphones is 2.14GB. From the log, we find that the available memory is 488MB on average, which is only 22.8% of the physical memory (90% of users have less than 31.6% of total memory available). This is mainly from the memory threshold of the low memory killer, below which it terminates applications. The lack of free memory may freeze a mobile device frequently and degrade user experience. The average memory consumption of a controllable activity process is 55.4MB in background and 116MB in foreground. We depict memory consumption of 25 popular applications (5 applications in Game, Messaging, Browsing, Portal/Video, Social categories) in foreground and background in the top of Fig. 13. The memory consumption in background is almost half of that in foreground, so that a mobile device lacks available memory if many applications are running in background. The time averaged memory size from controllable activity processes in background is 325MB. Note that the mobile OS and system processes occupy 60% of physical memory on average.

Warm and cold launch latency: In the bottom of Fig. 13, we present warm and cold launch latencies for the popular applications measured from our controlled experiment using Samsung Note2. To quantify the launch latency, we first measure the time durations until (1) screen rendering, and (2) loading application data in memory is completed, by filtering and monitoring Android logcat debugging outputs [33]. All other applications are unloaded before each measurement. We then regard the maximum of these two time durations as the launch latency. The average warm and cold launch latencies are 0.9s (rendering: 0.7s, memory loading: 0.4s) and 4.5s (rendering: 3.61s, memory loading: 3.58s), respectively. The game applications show the most drastic difference in latency, where the warm and cold launches take 12.6s and 1.8s, respectively. This is mostly due to loading high volume of texture data onto memory and rendering initial game scenes. For the tested popular applications, application preloading that transforms a cold launch into a warm launch decreases the start-up latency by 80% (3.6s).

User survey: We summarize key results from our survey. We first asked participants to choose major problems in their smartphones. 71% of participants chose short battery lifetime and 40% of them chose frequent freezes. Also, 46% of participants experience inconvenience from long start-up latency at least once a week. The battery lifetime that participants experience when it is fully charged is 9 hours on average, where it ranges from 3 to 24 hours. To increase battery lifetime and mitigate freezes, 82% of participants manually terminate applications and 28% of them use application killer software (e.g., Advanced Task Killer [34]). We also requested...