Context-aware Application Scheduling in Mobile Systems: What Will Users Do and Not Do Next?

Joohyun Lee
ECE, OSU
Columbus, OH
lee.7119@osu.edu

Kyunghan Lee *, **Euijin Jeong, Jaemin Jo**
ECE, UNIST
Ulsan, South Korea
{khlee,ujjeong,jaemin}@unist.ac.kr

Ness B. Shroff
ECE and CSE, OSU
Columbus, OH
shroff.11@osu.edu

ABSTRACT

Usage patterns of mobile devices depend on a variety of factors such as time, location, and previous actions. Hence, context-awareness can be the key to make mobile systems to become personalized and situation dependent in managing their resources. 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 context-dependency in application usage patterns, for which we classify contexts in a personalized manner with unsupervised learning methods. Using the knowledge acquired, we develop a novel context-aware application scheduling framework, CAS that adaptively unloads and preloads background applications in a timely manner. Our trace-driven simulations with 96 user traces demonstrate the benefits of CAS over existing algorithms. We also verify the practicality of CAS by implementing it on the Android platform.

ACM Classification Keywords

D.4.1 Operating Systems: Process Management—*Scheduling*; D.4.8 Operating Systems: Performance—*Modeling and prediction*

Author Keywords

Context-aware computing; Application unloading/preloading; Start-up latency; Energy minimization

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 fresh contents) have intrinsic tradeoffs that make it difficult to optimize simultaneously.

*indicates corresponding author.

UbiComp '16, September 12–16, 2016, Heidelberg, Germany
©2016 ACM. ISBN 978-1-4503-4461-6/16/0. . . $15.00
DOI: http://dx.doi.org/10.1145/2971648.2971680

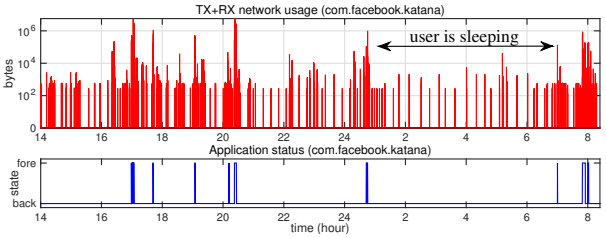


Figure 1. Daily network usage of the Facebook app and its corresponding state either being in the foreground or background.

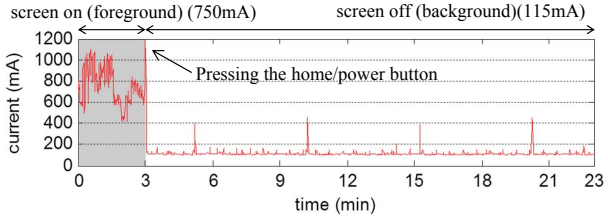


Figure 2. Measured power consumption of a popular game application for foreground and background states in a Galaxy Note 2 smartphone.

In a large-scale measurement study of 2000 Galaxy S3 and S4 devices by Chen et al. [9, 10], *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 [25, 28]. 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 usage[2] of a Facebook application on a Galaxy S7 smartphone running Android 6.0.1 in Figure 1, where the update or collection intervals are less than 20 minutes.[3] 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

[1]It is well known that frequent network traffic incorporates large ramp and tail energy overheads [11, 16].
[2]We log network usage by reading `/proc/uid_stat/[uid]/`.
[3]The authors in [28] revealed that the Facebook application uses network data every 5 minutes or every 1 hour in their large scale measurement between Dec 2012 to Nov 2014. They also revealed that network traffic from background applications consumes 84% of total network energy, mainly due to periodic contents updates and their tail energy consumption.

indeed observed in a popular game application, as shown in Figure 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.

To our knowledge, the most widely used application controller in Android [3] and iOS [5] 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 [19]. 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 [22, 24, 26, 31] 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 [10] 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. [10] 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 100 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 [7,27,29,30,33]. 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.

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 [7,12,14,27,29,30,33] 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. [14] 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 [6, 21, 23, 34, 35] 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 [3] and iOS [5] 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 [19]. Android version 6.0 (Marshmallow), released in October 2015, adopts new features called *App standby* and *Doze* mode [4] 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 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 [10] 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 α. Another algorithm, HUSH increases the suppression in-

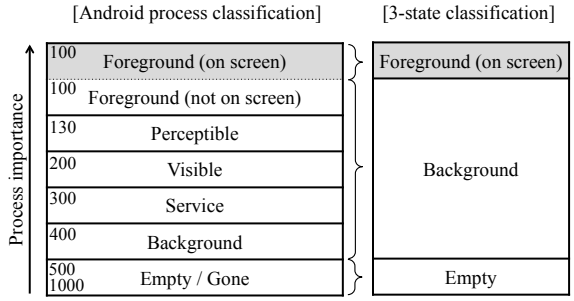


Figure 3. The process states defined in Android [1] (left) and our simplified three states (right).

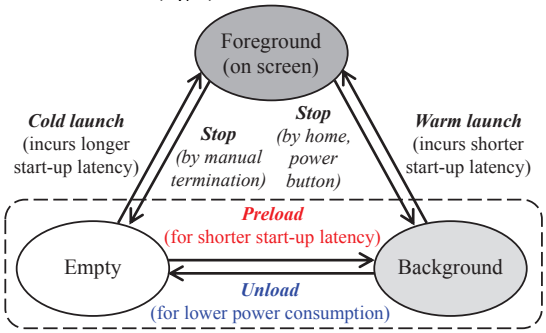


Figure 4. Our simplified states and transitions between a pair of states.

terval 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 σ). 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.

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 [1]. An Android application[4] installed on a device stays in one of the states at a time slot. Figure 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[5] 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[6] that are designed to be running all the time, and user-interactive applications[7] that are usable even without visible UIs.

[4]We interchangeably use process and application.
[5]The empty state corresponds to *suspended* in iOS [5].
[6]representatively phone caller and application launcher.
[7]applications such as music, radio and recorder.

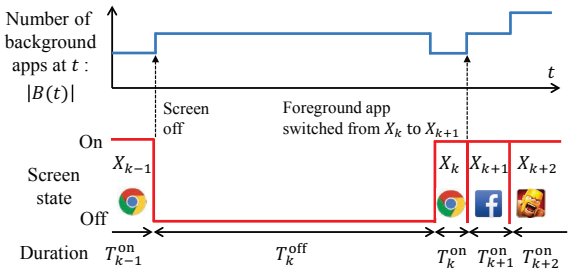


Figure 5. Our slotted time model of off and on periods (bottom) and an example of corresponding sets of background applications at each slot by LMK (top).

We depict the transitions between states in Figure 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. The k-th off period, T^{off}_k denotes a continuous screen-off duration when all applications are either in background or empty, while the k-th on period, T^{on}_k denotes a continuous screen-on time duration for which an application is being used in the foreground. Figure 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.

MEASUREMENT STUDY

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 1, and upload the data to our server daily. We anonymized all user information and IDs at the level of user devices. The valid data spans 1057 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 technical report [20].

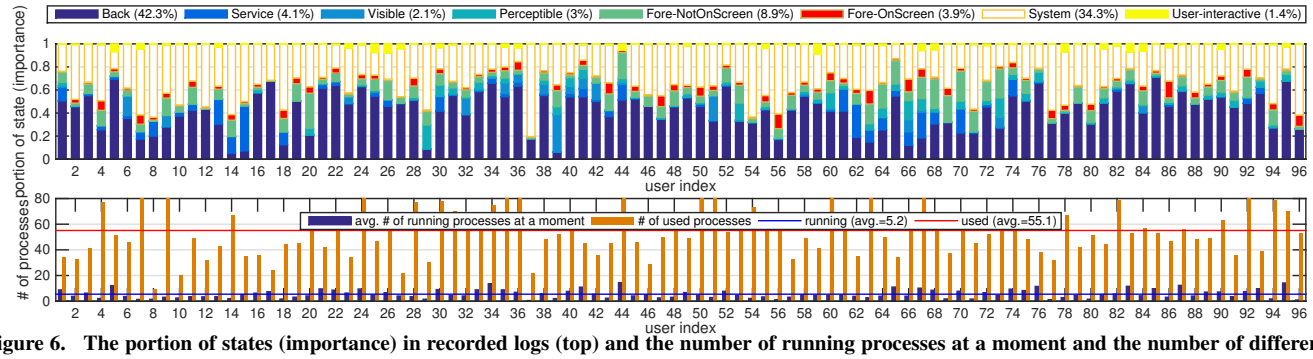


Figure 6. The portion of states (importance) in recorded logs (top) and the number of running processes at a moment and the number of different used processes (bottom). The numbers (-%) in the top graph indicate the average portion of each state across all participants (This figure is best viewed in color).

Table 1. Logged Events and Associated Fields.

Event Name	Associated Fields	Periods
Applist	List of all installed applications	-
Running apps	List, Importance, Memory usage	10 secs
Battery status	[Full, Charging, Not Charging], 0-100%	"
Screen status	[On, Off], Brightness (0-255)	"
Available memory	Memory in MB	"
Location	Longitude, Latitude, Accuracy	5 mins

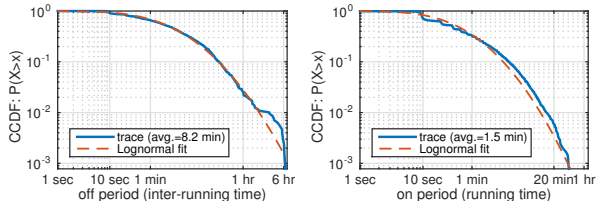


Figure 7. The CCDF (complementary cumulative distribution functions) of off (left) and on (right) periods of one user and the corresponding log-normal fittings.

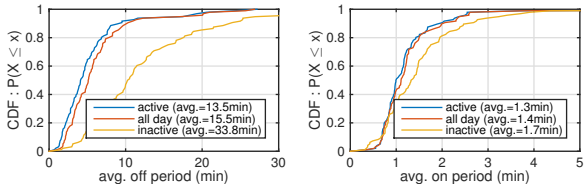


Figure 8. The CDF of average off (left) and on (right) periods of users.

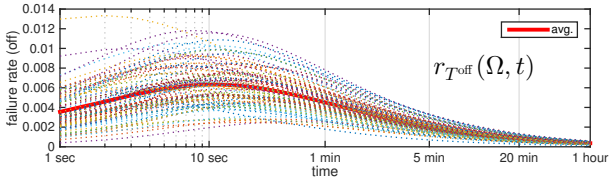


Figure 9. Off failure rates of users. The dotted lines are failure rates of each individual user.

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.

Key observations from the measurements

We summarize key observations in this subsection. More details are in the report [20].

Application usage statistics of users and states: In Figure 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. 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).*

Off/on period distribution: In Figure 7, we plot the typical off/on period distributions of a randomly chosen participant and show that the distributions are heavy-tailed. We verify by Cramer-Smirnov-Von-Mises (CSVM) [13] and Akaike [8] tests that off/on periods of all users have the best fit with *log-normal distributions*[8] 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 Figure 8. 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 $r_T(t) \triangleq \frac{f_T(t)}{1-F_T(t)}$, 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^{off} or T^{on}. We call off failure rate (from off to on) for T^{off} and on failure rate (from on to off) for T^{on}.

[8]The probability density function (PDF) of the log-normal distribution with parameters μ and σ is $(x\sigma\sqrt{2\pi})^{-1}\exp(-(\ln(x)-\mu)^2/2\sigma^2)$.

Table 2. 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.

category	process name	launches	time	top-1	top-2	top-3
Messaging	com.**kakao.talk**	27%	47s	**44%**	**25%**	7.3%
Browsing♯	com.android.**browser**	6%	161s	9.4%	9.4%	**10%**
Portal	com.**nhn.android.search**	4.4%	123s	2.1%	5.2%	9.4%
Browsing♯	com.sec.android.**app.sbrowser**	3.8%	129s	5.2%	5.2%	5.2%
Social	com.**facebook**.katana	3.3%	126s	-	6.3%	6.3%
Contacts♯	com.android.**contacts**	2.7%	20s	4.2%	1%	5.2%
Social	com.**nhn.android.band**	2.2%	49s	-	5.2%	1%
Browsing	com.android.**chrome**	2.2%	119s	3.1%	4.2%	4.2%
Setting♯	com.android.**settings**	1.8%	29s	-	1%	-
Social	com.**nhn.android.navercafe**	1.5%	82s	-	1%	2.1%
Game	com.supercell.**clashofclans**	1.5%	281s	-	2.1%	2.1%
Messaging	jp.**naver.line**.android	1.2%	29s	2.1%	-	2.1%

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

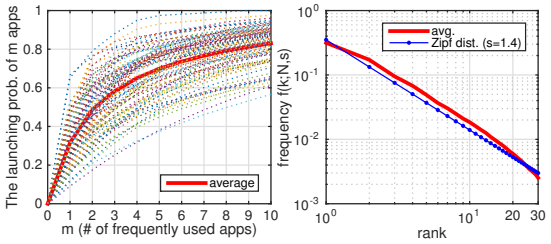


Figure 10. The CDF of the launching probability of m most frequently used applications (left) and Zipf distribution fitting for the average launching probability (right). The frequently used applications of each user are not identical. The dotted lines are for each individual user.

In Figure 9, we plot off failure rates, for each user (dotted lines) and on average (solid line). See [20] for graphs of on failure rates (omitted for brevity). 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* [17]. This indicates that *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 as the failure rate starts to get reduced.* This also suggests that the increasing backoff mechanisms of HUSH [10] and Doze [4] can be effective although their schedules are neither optimized nor personalized given that the individual failure rates (dotted lines in Figure 9) show distinct characteristics for different users.

Frequently used applications: In Table 2, 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 Figure 10, 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[9] 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.

Warm and cold launch latency: We quantify the application launch latency by measuring the maximum of the time durations until (1) screen rendering is finished, and (2) loading application data in memory is completed, by filtering and monitoring Android logcat debugging outputs [2]. For the 25 most popular applications tested (see [20]), *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.*[10] Thus, application preloading that transforms a cold launch into a warm launch can decrease the start-up latency by about 80%.

Context dependency: We also analyze app usage patterns incorporated under various contexts. Here, contexts correspond to any information that characterizes the situation of users, which enables us to predict future app/component invocations more accurately. In Figure 8, the average inter-launching time at inactive hours (24:00 to 9:00) is about 3.6 times longer than the average inter-launching time at active hours. In Table 2, we find that the average on periods are vastly different across applications (e.g., long running time for games and browsers, and short running time for messengers). We also observe other context dependencies such as previously used app (X_{k-1}) and the duration of the previous intervals (T^{off}_{k-1}, T^{on}_{k-1}), but they are omitted for brevity. For example, after using a messaging app, the next inter-launching times are typically shorter than the average, as the recipient of a message may respond quickly.

SYSTEM ARCHITECTURE

In this section, we propose our system design of CAS as depicted in Figure 11. Our framework consists of three major components: 1) context monitor, 2) user profiler, and 3) background application controller. Over these system components, CAS runs in three phases: *collection*, *pre-computation*, and *control*, each of which is operated as follows.

Collection phase and context classification: In the *collection* phase, context monitor collects various contextual information in background, to build information base on application usage pattern. Contexts we collect are screen state, time and location information, memory and CPU/network usage, application launch sequence, and battery level. Using the information base, the user profiler analyzes per-application usage behavior, cross-correlations of application usage behaviors, and resource (power/memory) consumption of background applications according to component-wise power models in [18,36], with diverse statistical measures such as failure rates and launching probabilities. Collection of data for learning may take some time (e.g., one week) in order to prepare a reasonable amount of statistics at first (e.g., when a user buys a new phone).

[9]The frequency of elements of rank k, $f(k;s,N)$ of a population of N applications is proportional to k^{-s}.

[10] The average of rendering or memory loading is shorter than launch latency, as the maximum of both is taken.

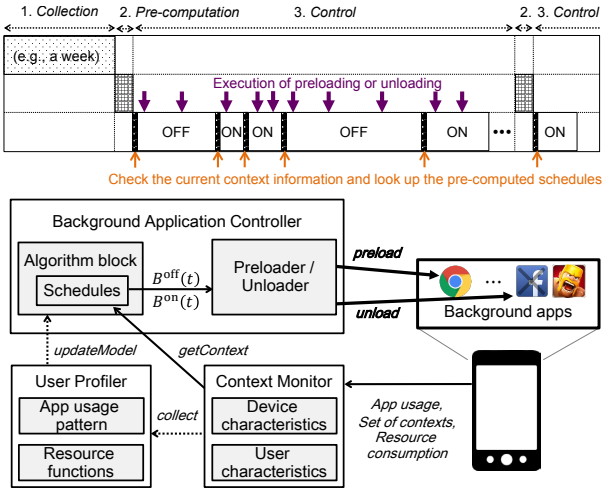


Figure 11. The overall architecture of CAS and its operations over

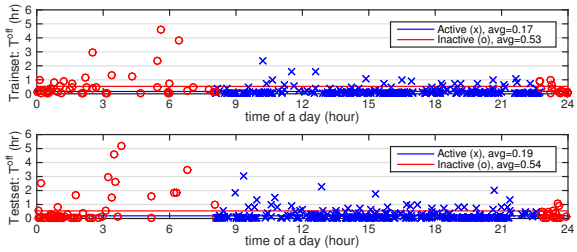


Figure 12. A sample off-period classification by "active" and "inactive" hours obtained from the first week trace (train set, top) of one user. The same classification is applied to the second week trace (test set, bottom).

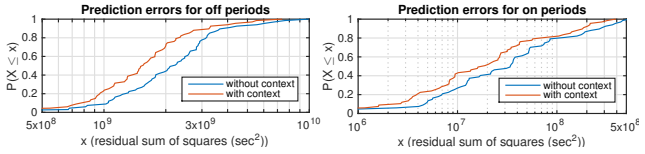


Figure 13. CDF of prediction errors for off (left) and on (right) periods of users.

To better exploit contexts, the collected context information of traces will be classified into several categories (e.g., Active/Inactive for time information), where the context information at a moment is defined as a tuple of several contexts (e.g., (*time*=Active, *last app*=Facebook)). People have different lifecycles and habits, so that contexts should be classified *individually*. Given that obtaining the labels for some contexts such as active hours from user input is intrusive to the user experience, automatic labeling is challenging. This challenge brings an unsupervised learning framework to our system, where we want to cluster (i.e., label) a set of samples with similar characteristics in the same category. Here, similarity can be quantified by a square error metric, which is widely used in literature. For instance, to simply classify *"time of a day"* information into several continuous time blocks, we solve the following problem to minimize the residual sum of square errors, by using a k-means algorithm [15].[11]

$$\min_{H_A} \sum_{k:S_k \in H_A} (T_k - \mathbb{E}[T_k | S_k \in H_A])^2 + \sum_{k:S_k \notin H_A} (T_k - \mathbb{E}[T_k | S_k \notin H_A])^2,$$

where S_k is time of a day of k-th sample (at the beginning), and H_A is the active timezone that is continuous (e.g., $H_A = [a, b]$, where a is 10:00 and b is 21:00). $S_k \in H_A$ if S_k is within the time interval of H_A. We depict an example of *"time of a day"* classification for off periods in Figure 12. The diurnal pattern of this user is clearly identified by our classification. We find that most of the users need only 2 continuous time blocks to explain their temporal activities and show very little gain from further separation.

In Figure 13, we depict the residual sum of squares of users for off and on periods in the test set (i.e., the second week of the trace), where the contexts (time of a day, previous off and on periods, last used application) are trained from the first week of the trace. We note that these dependencies including diurnal patterns are vastly different among users depending on their usage patterns and lifestyles. The residual sum of squares is decreased by 29.8% and 40.3% for off and on periods on average, respectively. Thus, it is clear that our automatic context classification results in more accurate prediction. We also find that the entropy[12] of the next launching app, X_k, is substantially reduced as well, omitted for brevity. Intuitively, lower entropy means reduced uncertainty and better predictability.

Pre-computation and control phases: Based on this analysis, the background application controller computes sets of background applications for possible combinations of contexts in both off and on periods, in the *pre-computation* phase. For each cluster C (i.e., a tuple of several contexts), we use the conditional distribution $T_k|C$ and conditional probability of $X_k|C$ where these conditional values are trained under the cluster C. In other words, our algorithm (which will be explained in the next section) will run for each cluster C. Therefore, we exclude the conditional information C in the rest of the paper, i.e., $T_k = T_k|C$ and $X_k = X_k|C$, for simplicity. This pre-computation happens once in a while (e.g., one time per week) to adapt for the change of the application usage behavior. Pre-computation also runs during the inactive hours with the device connected to its charger, to avoid any inconvenience to users. During the *control* phase, at the start of each off or on period, the controller calls the context monitor and acquires the contextual information at the moment as its input. Based on the pre-computed list of background applications at each moment for the given contextual information, background application controller executes pre/unloading during the on or off period. As the recommended list of background applications at each moment are pre-computed, these executions do not bring any computational burden.

ALGORITHM DESIGN

In this section, we formulate a submodular optimization problem that selects the best set of background applications to minimize the total penalty in energy and start-up latency. Then, we develop a practical scheduling algorithm for CAS.

System model

System states: We let Ω ($|\Omega| = N$) denote the set of controllable applications of a user, which does not include any system and user-interactive processes. We define $B(t) \subseteq \Omega$ and $\bar{B}(t) \subseteq \Omega$ to be the sets of applications in the background and empty states, respectively, which are our control knobs.

[11] We similarly formulate and solve this problem for other contexts such as previous off and on periods.

[12] The entropy for the application is $-\sum_{i \in \Omega} \mathbb{P}[X_k = i] \log_2 \mathbb{P}[X_k = i]$.

We define the system state as an off/on period as in Figure 5, where the off period denotes a continuous duration when all applications are either in background or empty, while the on period denotes a continuous time duration for which an application is being used in the foreground. We denote T_k^{off} and T_k^{on} as random variables of the k-th off and on period, respectively. We denote X_k as the foreground application that runs during the time duration of T_k^{on}.[13] We recall that the failure rate is defined as $r_T(t) \triangleq \frac{f_T(t)}{1-F_T(t)}$, for t such that $F_T(t) < 1$. T can be either T^{off} or T^{on}. We also define the partial failure rates for a set of empty applications $\bar{B}(t)$ as $r_T(\bar{B}(t),t) = r_T(t) \cdot \mathbb{P}[X \in \bar{B}(t)]$, which quantifies the rate that one of empty applications $\bar{B}(t)$ is launched at t.

Power consumption and memory model: For applications included in Ω, we define a power function, $P : 2^{\Omega} \to \mathbb{R}_+$ and a memory function $M : 2^{\Omega} \to \mathbb{R}_+$ that respectively represent the amount of the average power and memory consumption of a set of background applications. Based on the observations made in [36], we model P as a submodular function[14] for any $B(t)$, as applications share hardware components, and they become more power-efficient as the utilization becomes higher. A memory function M is a linear additive function[15] for any $B(t)$. Note that we use average power and memory consumption for long-term optimization. For simplicity, we assume that the energy consumption for preloading and unloading is virtually negligible and that the transition delay is much shorter than one time slot. These assumptions practically make sense as the preloading/unloading consume its power less than a few seconds and happen only a few times an hour. More detailed discussion on the consumption of the transition energy will be provided in our simulation results.

Problem formulation

We aim to develop a scheduling algorithm for CAS that reduces and balances energy consumption and user disutility from application launch latency, under a given memory budget M_{th}. Since there is a trade-off between the energy consumption and the user disutility (i.e., cold launch probability), we adopt a parameter γ to treat both metrics as a unified measure. We let a user who is less sensitive to latency but is keen to extend battery lifetime choose a smaller γ value, and vice versa. The optimal scheduling algorithm for CAS can be obtained from the optimization problem that minimizes both the energy consumption and the user disutility over the infinite time horizon. The optimization problem can be disjointly split into off-period optimization and on-period optimization. We present the formal formulation of the off-period optimization as follows and omit the on-period formulation for brevity.

$$\min_{\substack{B^{\text{off}}(t),\forall t:\\ M(B^{\text{off}}(t))<M_{\text{th}}}} \sum_{t=1}^{\infty} \mathbb{P}[T^{\text{off}}=t]\Big(\sum_{\tau=1}^{t} P(B^{\text{off}}(\tau)) + \gamma \cdot \mathbb{P}[X \notin B^{\text{off}}(t)]\Big)$$

$$= \min_{\substack{B^{\text{off}}(t),\forall t:\\ M(B^{\text{off}}(t))<M_{\text{th}}}} \sum_{t=1}^{\infty} \mathbb{P}[T^{\text{off}} \geq t]\Big(P(B^{\text{off}}(t)) + \gamma \cdot r_{T^{\text{off}}}(\bar{B}^{\text{off}}(t),t)\Big).$$

The summation of $P(B^{\text{off}}(\tau))$ from $\tau = 1$ to t indicates the energy consumption when the length of an off period is t and the second term quantifies the expected disutility from the cold launch of an application weighted by γ. By restating energy and disutility terms using $\mathbb{P}[T^{\text{off}} \geq t]$, we have the second line. Since we have assumed that the latency and energy overhead for preload and unload are negligible, the sets of optimal background applications for time slots and their resulting snapshot objectives become uncorrelated. Hence, in this formulation achieving the optimality in each snapshot (i.e., time slot) warrants the global optimality. In the rest of the paper, we omit the superscript off in T^{off}, $B^{\text{off}}(t)$, and $r_{T^{\text{off}}}(B^{\text{off}}(t),t)$ unless we need to emphasize them.

Scheduling algorithm design

If $P(B(t))$ were additive, then this problem becomes a **0-1** knapsack problem, which can be solved using dynamic programming. However, $P(B(t))$ is submodular in general because applications share power-consuming components. Then, this problem becomes a submodular minimization with a weighted cardinality constraint (i.e., memory constraint), which has been proven to be NP-hard in [32]. Hence, we propose a greedy-based algorithm with limited complexity (e.g., up to quadratic time complexity), which makes locally optimal choices in finding a set of background applications. This may result in sub-optimal performance but works well in practice due to the Zipfian distributed launching probability. Intuitively, most dominant or frequently used application with high launching probabilities are chosen as background applications in the first few iterations.

CAS scheduling algorithm

input: $P(\cdot), M(\cdot), r_T(\cdot), \mathbb{P}[X=x], \gamma, M_{\text{th}}$
output: $a_1, ..., a_N$, $c_1, ..., c_N$, and $B(1), ...B(t_{\max})$
Step (A) Compute a local optimal sequence.
1: $A_0 \leftarrow \emptyset$.
2: **for** $m = 1$ **to** N **do**
3: $\quad a_m \leftarrow \text{argmax}_{i \in \Omega \setminus A_{m-1}} \frac{\mathbb{P}[X=i]}{P(A_{m-1} \cup \{i\}) - P(A_{m-1})}$.
4: $\quad c_m \leftarrow \max_{i \in \Omega \setminus A_{m-1}} \frac{\mathbb{P}[X=i]}{P(A_{m-1} \cup \{i\}) - P(A_{m-1})}$.
5: $\quad$ **if** $M\big(A_{m-1} \cup \{a_m\}\big) > M_{\text{th}}$ **then**
6: $\quad\quad c_m \leftarrow \infty$ and **break**
7: $\quad$ **else** $A_m \leftarrow A_{m-1} \cup \{a_m\}$.
Step (B) Assign controls at each time slot.
1: **for** $t = 1$ **to** $t_{\max}$ **do**
2: $\quad m \leftarrow \max\{j | c_i \geq \frac{1}{\gamma \cdot r_T(t)}, \forall i \leq j\}$.
3: $\quad B(t) \leftarrow A_m$.

Note that $t_{\max}$ is the maximum duration from all observable off or on periods such that $\mathbb{P}[T > t_{\max}]$ goes to zero and $B(t) = \emptyset$ for $t > t_{\max}$. *Our scheduling algorithm pre-computes the entire sequence of locally optimal control actions in step (A) and assign them in each slot in step (B).* In step (A), if more than one application becomes tied, it breaks the tie by arbitrarily choosing one of them. The computational complexity of our algorithm is $O(N^2 + NT)$ where complexities of step (A) and (B) are $O(N^2)$ and $O(NT)$, respectively.

We note that our scheduling algorithm does not change its control decision if the environmental conditions (e.g., power/memory functions, failure rates, or launching probabilities) are maintained. As those conditions are stationary or slowly changing over time, re-computation of the algorithm happens rarely in practice.

[13] We omit the subscript k unless confusion arises.
[14] $P(A \cup B) \leq P(A) + P(B) - P(A \cap B)$ for $A, B \subset \Omega$.
[15] For any disjoint sets $A, B \subset \Omega$, $M(A \cup B) = M(A) + M(B)$.

Table 3. Summary of contextual information.

Info	Last used app (X_{k-1})	Time of a day (S_k)	Previous durations ($T^{\text{off}}_{k-1}, T^{\text{on}}_{k-1}$)
Class	$1, \cdots, N$	Active, Inactive	Short, Long

TRACE-DRIVEN SIMULATION

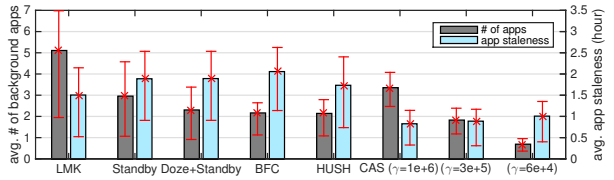
Setup: To evaluate power consumption and latency performance of CAS for our measurement traces, we develop a trace-driven simulator incorporating the average power and memory functions, $P(\cdot)$ and $M(\cdot)$. We model $P(\cdot)$ by the component-wise power model (e.g., CPU, screen, WiFi, cellular, and GPS) in [36] and our measurement on utilization of components for each application in our traces. $M(\cdot)$ is directly computed from our measurement log. In the trace-driven simulation, we compute the performance of CAS in which the control decisions are made by the proposed scheduling algorithm. All statistics and classifications are obtained from the first week of the trace (i.e., training set) and simulations are conducted for the second week of the trace (i.e., test set). We further compare a set of existing algorithms including the default Android scheduler (LMK), App standby and Doze mode [4] in Android 6.0, BFC and HUSH proposed in [10] with CAS. The contextual information we used is summarized in Table 3.[16] As the gain from location information is turned out to be negligible, we exclude the location information. Authors in [23] also found that the benefit from location information in prediction accuracy is minimal as it is already partially captured by the application sequence and time information. The parameters of BFC ($\alpha = 0.1$) and HUSH ($\sigma = 1.2$) are chosen as in [10]. The memory threshold for CAS is set to be 30% of the total memory size for each user leading to 840MB on average.[17]

Key Results

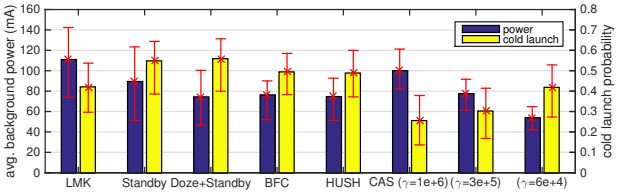
We depict the performance of different scheduling algorithms in Figure 14, and summarize results as follows.

Inefficiency of LRU-based LMK: As a baseline, we evaluate the performance of LMK from our experimental logs. *The average power consumption from background applications is about 111mA,* which is much higher than the typical idle power consumption of 10mA in the most up-to-date smartphones. *The average power consumption of a foreground application during screen on periods is about 562.5mA.* Given that, our experimental traces show that the energy consumption of background applications amounts to 48.5% of the total battery capacity under LMK. These measurements lead to 12.2 hours of average battery life for the devices in our experimental logs whose average battery capacity is about 2800mAh. The average cold launch probability with LMK is measured to be 43% with average memory occupancy of 212MB from controllable background applications.[18]

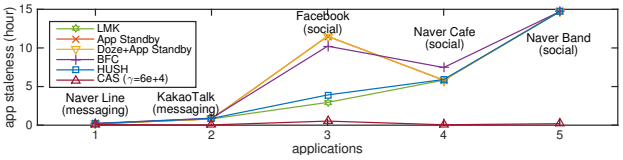
Android 6.0, BFC and HUSH [10]: The new feature, *App standby* [4], in Android 6.0 unloads applications that have no foreground activity for more than 3 days. The portion of



(a) Number of background applications and staleness.



(b) Background power and cold launch probability.



(c) Staleness of social and messaging apps on one user.

Figure 14. Comparison of scheduling algorithms. The error bars indicate 25th and 75th percentiles.

unique applications that can be affected by *App standby* option is observed to be 73% of total installed applications, but only 39% of background activities under LMK are affected (See Figure 14(a)). This is due to active killing of applications under memory pressure in LMK. Therefore, energy saving for background applications over LMK is limited to 19.4% (total energy saving is 9.4%). Another feature, *Doze* mode [4] in Android 6.0, restricts background activities is enabled after a user leaves the device for an hour. Then, the suppression time windows are increasing as 1, 2, 4, and 6 hours, where the maintenance windows are scheduled in between suppression windows for 5 minutes. *Doze* mode further reduces background energy by 33% over LMK, but its additional energy saving is not significant as the time portion of off periods over an hour is only about 28%. Note that only 3% of off periods are longer than an hour.

BFC and HUSH algorithms unload background activities more aggressively each of which suppresses 51% and 47% more compared to LMK. The background (total) energy savings over LMK in BFC and HUSH are 31.2% (15.1%) and 32.7% (15.8%), respectively. One potential problem of BFC and HUSH is that they update control decisions only by relying on latest activities of applications. For example, in HUSH, the suppression interval is reset to an initial value (i.e., 1 min) every time an application gets a foreground activity, which is not efficient for low active applications. In all these schemes, their cold launch probabilities and staleness are higher than LMK, since these algorithms only suppress background activities.

CAS: CAS achieves diverse operating points depending on the trade-off parameter γ. Our scheduling loads more background applications as γ increase as shown in Figure 15. Also, according to our finding that the launching probability decreases as the elapsed time passes by in each off/on period

[16] We used the k-means algorithm to classify "time of a day" and "previous durations".

[17] Our measurement data indicates that on average about 60% of total memory is occupied by the OS and system processes.

[18] Note that 212MB is only for the background applications. In general, the memory utilization of a device is much higher as it further involves foreground and system processes.

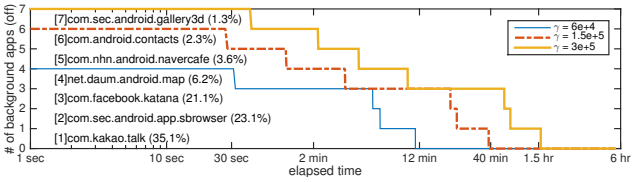


Figure 15. Background application schedules of CAS for one user in off periods. The x-axis is in log scale. We list the application names of the ordered sequence in the graph and the numbers (-%) indicate the launching probabilities. 1: Messaging, 2: Browsing, 3, 5: Social, 4: Navigation, 6: Contacts, 7: Utility.

(i.e., *negative aging*), CAS unloads more and more background applications as time goes by. As the failure rate starts to decrease after around 30 seconds, low priority applications are unloaded sequentially. Eventually, all background applications are unloaded after 12, 40, and 90 minutes for different operating points, $\gamma =$ 6e+4, 1.5e+5, and 3e+5, respectively. CAS achieves a similar cold launch probability with only 0.7 background applications (14% of LMK) for $\gamma =$ 6e+4 in Figure 14. The background and the total energy savings over LMK become as high as 51% and 25% on average for γ =6e+4. Also, CAS reduces the cold launch latency by 26% over LMK for γ=1e+6, with lower energy consumption.

Staleness: We define *app staleness* as the average elapsed time since the last background or foreground activity of an application as in [10]. This metric captures the user experience especially for applications that need to regularly update their contents (e.g., social networking and messaging applications). The average app staleness under LMK is 1.5 hours. In Android 6.0, BFC and HUSH, their app staleness values are always higher than or equal to LMK because these algorithms do not restore unloaded background activities as shown in 14(a). Unlike other algorithms, CAS preloads background applications and reduces the average staleness by 33% and 41% over LMK for $\gamma =$ 6e+4 and 3e+5, respectively.

To see how the app staleness varies with scheduling algorithms, we compare app staleness of five popular social and messaging applications from different scheduling algorithms for one user in Figure 14(c). Because this user only infrequently uses the Facebook app and sometimes does not use it for more than 3 days, the App standby of Android 6.0 unloads it, which in turn leads to a very high app staleness. However, CAS predicts the moment that an application is used next, so that the average staleness becomes much shorter than other algorithms across all applications including the social and messaging applications.

Energy overheads of CAS: There are energy overheads in CAS that are from logging information, processing algorithms, preloading actions and wakeup alarms for pre/unloading. Note that the application controller sleeps during the time when the set of background applications stays the same, and wakes up only when the control action is needed to make changes. Our measurement reveals that contextual information logging without location information consumes only about 5mA. The *pre-computation* phase consumes 30mAh to compute the control policies for all context sets. This corresponds to power consumption of 1.25mA under a daily update frequency. Wakeup alarming for a control action and actual preloading of an application are turned out to consume on average about 0.03mAh and 0.4mAh, respectively. Note that unloading happens in a flash, and thus it consumes nearly 0mAh. In CAS, the average frequencies of wakeup alarms and preload actions are less than once in 2 mins for the chosen parameters. Overall, the energy overhead required for running CAS does not exceed 23.3mAh per hour at maximum, which is about 0.8% of the battery capacity and is much smaller than the huge gain obtained from CAS. We take into account these energy overheads in CAS.

Table 4. Comparison of scheduling algorithms.

Scheduler	Power♯	Cold launch	Memory♯	Lifetime†	Latency‡
LMK-LRU	111mA	43%	212MB	12.2 hr	2.45sec
App Standby	89.5mA	54.8%	131MB	13.5 hr	2.87sec
Doze+Standby	74.4mA	55.9%	103MB	14.5 hr	2.91sec
BFC [10]	76.4mA	49.4%	93MB	14.4 hr	2.67sec
HUSH [10]	74.7mA	48.9%	91MB	14.5 hr	2.66sec
CAS (γ=1e+6)	100.1mA	25.6%	142MB	12.8 hr	1.82sec
CAS (γ=6e+4)	53.9mA	41.9%	31MB	16.3hr	2.41sec
Oracle	10mA	0%	0MB	21.9 hr	0.9sec

♯: Power/memory consumption of background applications including energy overhead. The voltage ranges from 3.7V to 3.8V. The idle background power is 10mA.
†: Based on 21% of screen on periods and 2800mAh of battery.
‡: Based on 4.5sec of cold launch and 0.9sec of warm launch latencies.

Battery life and start-up latency: To quantify the gain in the user-perceived metrics such as battery lifetime and expected start-up latency, we calculate them based on our measurement over popular applications and summarize them in Table 4. We include a simulation of the ideal yet infeasible scheduler, *Oracle*, that exactly knows when and which application the user will use next. The average battery lifetime of a device is extended to 16.3 hours in CAS (γ=6e+4) from 12.2 hours observed under LMK. Note that the upper bound of battery lifetime simulated from *Oracle* is 21.9 hours. By modifying the parameters, the expected start-up latency is reduced from 2.8sec in LMK to 1.82sec in CAS (γ=1e+6), with a similar lifetime as LMK.

ANDROID IMPLEMENTATION

We implement CAS on Galaxy Note 2 (the most popular device in our traces), which runs Android 4.4.2, KitKat. Three major components (context monitor, user profiler, and background application controller) are implemented and packaged as a system service. Our implementation of context monitor uses methods defined under *SensorEventListener* interface of Android such as *onSensorChanged* and *onAccuracyChanged* to minimize the energy consumption and uses delayed write for saving data in the SQL database (i.e., SQLite of Android) in a highly energy efficient manner. To this end, the background application controller uses *BroadcastReceiver* and *AlarmManager* to execute preloading and unloading at desired moments with little use of CPU resource. In order to realize preloading, we use *getLaunchIntentForPackage* method together with *startActivity* included in *PackageManager* of Android.

To unload processes, we implement the linux shell command execution of *am force-stop <Package Name>*[19] in Android using the Android NDK (native development kit), where *am* stands for activity manager. To make CAS work independently from Android LMK or Linux OOM (out of memory)

[19]We gain the super user (*su*) access by rooting the device to perform *am*, which will be unnecessary once our scheduling algorithm is integrated with Android.

Table 5. Experimental results of CAS and LMK over 1-day traces of two users on Galaxy Note 2.

User index	User 46		User 61	
Number of installed apps	37		64	
Portion of screen on periods	8.1%		22%	
Daily screen on durations	1.9 hours		5.3 hours	
Avg. off period	16.2 mins		16.9 mins	
Avg. on period	0.85 min		0.9 min	
Scheduling	LMK	CAS	LMK	CAS
Avg. # of background apps in run	6.0	0.2	4.5	0.99
Avg. screen-off power (mA)	72	26.7	100	44.7
Avg. screen-on power (mA)	388	378	526	460
Avg. power (mA)	97.5	55.8	194	136
Expected lifetime‡ (hour)	31.8	55.5	16.0	22.8

‡: Battery size is 3100mAh.

killer underneath Android platform without interfering with them, we substantially relaxed all low-memory related parameters and virtually disabled such resource schedulers. In order to let an application stay unloaded as per our decision, we also intercept app invocations[20] such as the synchronous RPC (Remote Procedure Call) mechanism called *Binder* and asynchronous IPC (Inter-Process Communication) message passing mechanism called *Intent*, which can wake up an unloaded process. This may delay notifications or messages of unloaded processes which we will discuss later.

Android Experiment

For the experiment of our platform, our service is designed to precisely follow the application and screen behaviors precomputed over a collected trace as a time series for each scheduling algorithm. Note that we choose to perform this replay style emulation as it is better than a hand-carried experiment from the perspective of ensuring a fair comparison between the algorithms. Our replay service turns on and off screen by using *WakeLock* method in *PowerManager* class and *lockNow* method in *DevicePolicyManager* class, respectively. Because our experiment rules out any human intervention, it is reasonable to keep the system awakened using *WakeLock* while we emulate a screen on period. Although our emulation method is not perfect in mimicking user behaviors in foreground UI such as touch actions, it is fair to say that this end-to-end evaluation capturing all possible system overheads sets a baseline of the performance of CAS in reality.

We summarize the results of CAS (γ=6e+4) and LMK and the usage patterns for the two randomly chosen users in Table 5. One of these users turned out to be a light user and the other a mild to heavy user. We find that the energy savings of CAS from LMK are 43% and 30% for each user, as the numbers of background applications in run are significantly reduced. The average power during screen on periods is also reduced since CAS unloads background applications both in screen off and on periods. The experimental results confirm that energy saving from CAS can be indeed significant in practice. As a future work, we plan to extend our experiment toward user studies that involve evaluations of user-perceived benefits with CAS installed in the actual user devices.

DISCUSSIONS AND CAVEATS

There are practical issues that need to be considered before CAS can be widely used. The issues are mostly on the application characteristics and semantics that can be affected by controlling the application.

[20]This interception is similarly implemented as [10].

Discomfort from Unloading: Our application control may delay notifications or messages for unloaded applications. This can have both pros and cons, as deferring or neglecting advertisement messages can relieve one's stress and receiving important messages later can be big losses. However, long delay occurs only when these applications are not likely to be launched for a long time, so that the actual inconvenience may not be critical, as our result on *staleness* confirms. To avoid such inconvenience, we can consider *staleness* of applications in our objective directly, and allow background activities intermittently to reduce or bound *staleness*, which we leave as a future work. We can also whitelist some critical applications from application controlling as follows.

Whitelisting: Some apps should not be unloaded from background even though they may be infrequently used. For example, caller apps such as Skype should be able to receive calls at anytime. Thus, application categories such as call, music, radio, and recorder need to be excluded from application control, which we already excluded them as *user-interactive* processes. A more intelligent way is to ask users whether they want to *whitelist* infrequently used apps as in [31]. We can also use crowdsourced statistics[21] of applications to minimize the need for user inputs.

Privacy issue: Our context monitor and user profiler can have privacy sensitive data (e.g., sleeping hours from *time of a day* classification), which should be encrypted and only accessible by CAS. We note that because CAS runs on a mobile device and does not rely on cloud resources for analyzing its application usage patterns, there is no privacy or security concern for leaking personal data to Internet.

Dataset bias: Our trace data can be biased in the sense that we collected traces of the participants from Internet user communities in Korea, who may be more tech-savvy than general populations. We will collect more diverse trace data in the future work, and study the performance depending on the level of user acitivity.

CONCLUDING REMARKS

CAS requires sufficient application usage history with context information (i.e., long *collection* phase). For future work, we are interested in training model parameters faster using a learning framework, and crowdsourcing of statistics from the devices of the same type or of similar attributes, which can bootstrap the *collection* phase. We will study other energy-efficient contexts and how to classify them to increase the prediction accuracy. We are interested in designing a more efficient algorithm using other optimization methods, which can adapt the trade-off parameter γ to provide the best user experience as per the preference of the user.

ACKNOWLEDGEMENT

This 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, the Ministry of Science, ICT & Future Planning (NRF-2014R1A1A1006330), and by IITP grant funded by the Korea government (MSIP) (No. B0126-16-1064, Research on Near-Zero Latency Network for 5G Immersive Service).

[21]Individual choices will be anonymized and only the averaged statistics will be shared.

REFERENCES

1. Android Developers. 2015a. ActivityManager.RunningAppProcessInfo. (2015). **http://developer.android.com/reference/android/app/ActivityManager.RunningAppProcessInfo.html**.
2. Android Developers. 2015b. Android logcat. (2015). **http://developer.android.com/tools/help/logcat.html**.
3. Android Developers. 2015c. Managing Your App's Memory. (2015). **https://developer.android.com/training/articles/memory.html**.
4. Android Developers. 2016. Optimizaing for Doze and App Standby. (2016). **https://developer.android.com/training/monitoring-device-state/doze-standby.html**.
5. Apple Developer. 2015. App Programming Guide for iOS: The App Life Cycle. (2015). **https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html**.
6. Ricardo Baeza-Yates, Di Jiang, Fabrizio Silvestri, and Beverly Harrison. 2015. Predicting The Next App That You Are Going To Use. In *Proc. of ACM Web Search and Data Mining*.
7. Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. 2011. Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage. In *Proc. of ACM Human-Computer Interaction with Mobile Devices and Services (MobileHCI)*.
8. Kenneth P. Burnham and David R. Anderson. 2004. Multimodel Inference: Understanding AIC and BIC in Model Selection. *Sociological Methods and Research* 33, 2 (2004), 261–304.
9. Xiaomeng Chen, Ning Ding, Abiliash Jindal, Y Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015a. Smartphone Energy Drain in the Wild: Analysis and Implications. In *Proc. of ACM Sigmetrics*.
10. Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015b. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In *Proc. of ACM MobiCom*.
11. Ning Ding, Daniel Wagner, Xiaomeng Chen, Y Charlie Hu, and Andrew Rice. 2013. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In *Proc. of the ACM SIGMETRICS*.
12. Trinh Minh Tri Do, Jan Blom, and Daniel Gatica-Perez. 2011. Smartphone usage in the wild: a large-scale analysis of applications and context. In *Proc. of ACM conference on multimodal interfaces*.
13. W. T. Eadie, M. G. Roos, and F. E. James. 1971. *Statistical Methods in Experimental Physics*. Elsevier Science and Technology.
14. Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. 2010. Diversity in smartphone usage. In *Proc. of ACM MobiSys*. 179–194.
15. John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979), 100–108.
16. Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. A close examination of performance and power characteristics of 4G LTE networks. In *Proc. of ACM MobiSys*.
17. Jaeseong Jeong, Yung Yi, Jeong-woo Cho, Do Young Eun, and Song Chong. 2013. Wi-fi sensing: Should mobiles sleep longer as they age?. In *IEEE INFOCOM*.
18. Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. 2012. DevScope: a nonintrusive and online power analysis tool for smartphone hardware components. In *Proc. of ACM Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.
19. Jang Hyun Kim, Junghwan Sung, Sang Yun Hwang, and Hyo-Joong Suh. 2015. A Novel Android Memory Management Policy Focused on Periodic Habits of a User. In *Ubiquitous Computing Application and Wireless Sensor*. 143–149.
20. Joohyun Lee, Kyunghan Lee, Euijin Jeong, Jaemin Jo, and Ness B. Shroff. 2016. Context-aware Application Control in Mobile Systems: What Will Users Do and Not Do Next?, Technical report. (2016). *available at* **https://www.dropbox.com/s/857n6dd9gko5z2j/CAS-TR.pdf**.
21. Nagarajan Natarajan, Donghyuk Shin, and Inderjit S Dhillon. 2013. Which app will you use next?: Collaborative filtering with interactional context. In *Proc. of ACM Recommender systems (RecSys)*.
22. Adam J Oliner, Anand Iyer, Eemil Lagerspetz, Sasu Tarkoma, and Ion Stoica. 2012. Collaborative energy debugging for mobile devices. In *Proc. of USENIX Workshop on Hot Topics in System Dependability (HotDep)*.
23. Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M Marlin. 2013. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proc. of ACM UbiComp*.
24. Abhinav Pathak, Y Charlie Hu, and Ming Zhang. 2011. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proc. of ACM Workshop on Hot Topics in Networks (HotNets)*.
25. Abhinav Pathak, Y Charlie Hu, and Ming Zhang. 2012a. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proc. of ACM EuroSys*. 29–42.
26. Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff. 2012b. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of ACM MobiSys*.

27. Ahmad Rahmati, Clayton Shepard, Chad Tossell, Lin Zhong, and Philip Kortum. 2014. Practical Context Awareness: Measuring and Utilizing the Context Dependency of Mobile Usage. *IEEE Transactions on Mobile Computing* (2014).

28. Sanae Rosen, Ashkan Nikravesh, Yihua Guo, Z Morley Mao, Feng Qian, and Subhabrata Sen. 2015. Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild. In *Proc. of ACM Conference on Internet Measurement Conference (IMC)*.

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

30. Choonsung Shin, Jin-Hyuk Hong, and Anind K Dey. 2012. Understanding and prediction of mobile application usage for smart phones. In *Proc. of ACM UbiComp*.

31. Indrajeet Singh, Srikanth V Krishnamurthy, Harsha V Madhyastha, and Iulian Neamtiu. 2015. ZapDroid: managing infrequently used applications on smartphones. In *Proc. of ACM UbiComp*. 1185–1196.

32. Zoya Svitkina and Lisa Fleischer. 2011. Submodular approximation: Sampling-based algorithms and lower bounds. *SIAM J. Comput.* 40, 6 (2011), 1715–1737.

33. Hannu Verkasalo. 2009. Contextual patterns in mobile service usage. *Personal and Ubiquitous Computing* 13, 5 (2009), 331–342.

34. Yichuan Wang, Xin Liu, David Chu, and Yunxin Liu. 2015. EarlyBird: Mobile Prefetching of Social Network Feeds via Content Preference Mining and Usage Pattern Analysis. In *Proc. of ACM MobiHoc*.

35. Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast app launching for mobile devices using predictive user context. In *Proc. of ACM MobiSys*.

36. Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. 2012. AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring.. In *Proc. of USENIX ATC*.