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

†J. Lee is with the Department of Electrical and Computer Engineering at The Ohio State University, e-mail: lee.7119@osu.edu.

*‡K. Lee, the corresponding author, E. Jeong, and J. Jo are with the School of Electrical and Computer Engineering, UNIST (Ulsan National Institute of Science and Technology), South Korea, e-mails: {khlee, ujjeong, jaemin}@unist.ac.kr.

♯N. B. Shroff holds a joint appointment in both the Department of ECE and the Department of CSE at The Ohio State University, e-mail: shroff.11@osu.edu.

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. 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 1.17004.01).

A preliminary version of this work was presented at ACM UbiComp 2016 [1].

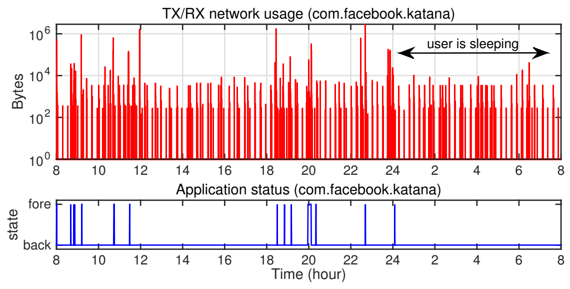


Fig. 1. Daily network usage of the Facebook app and its corresponding state either being in the foreground or background. The Facebook app incurs background network traffic even when the user is not interacting with it.

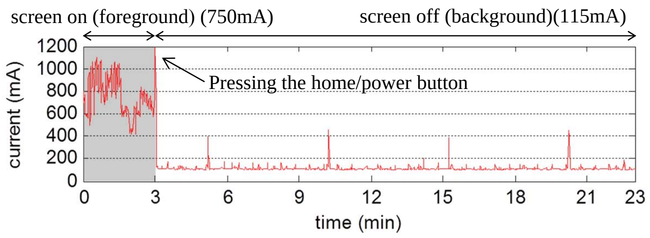


Fig. 2. Measured power consumption of a popular game application for foreground and background states in a Galaxy Note 2 smartphone.

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 usage[2] 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

[1]It is well known that frequent network traffic incorporates large ramp and tail energy overheads [6], [7].

[2]We log network usage by reading `/proc/uid_stat/[uid]/`.

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 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.

[3]The authors in [4] 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.

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.

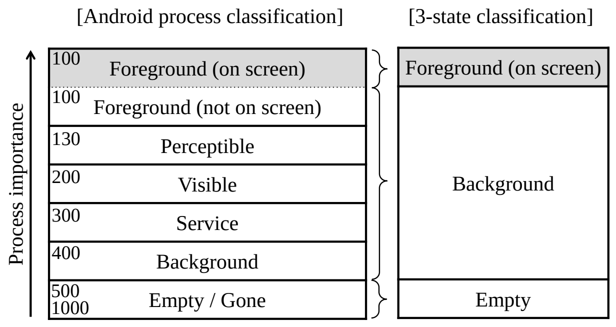


Fig. 3. The process states defined in Android [28] (left) and our simplified three states (right).

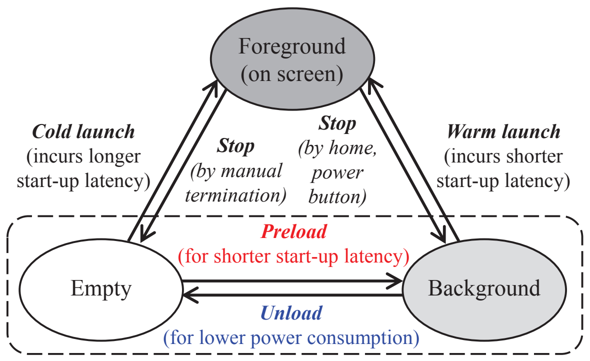

Foreground (on screen)
Cold launch (incurs longer start-up latency)
Stop (by manual termination)
Stop (by home, power button)
Warm launch (incurs shorter start-up latency)
Preload (for shorter start-up latency)
Empty
Background
Unload (for lower power consumption)

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

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 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 α. 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 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. 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[4] 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[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 (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

[4]We interchangeably use process and application.

[5]The empty state corresponds to *suspended* in iOS [9].

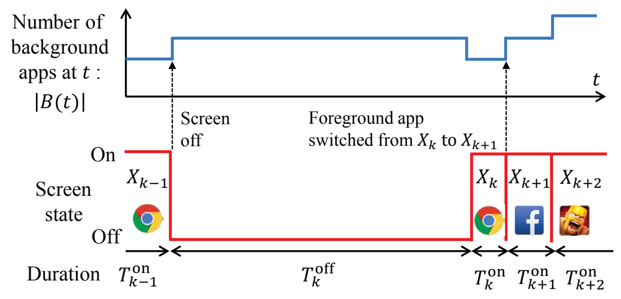


Fig. 5. Our slotted time model of off and on periods (bottom) and an example of corresponding sets of background applications at each slot by LMK (top).

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

TABLE I
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

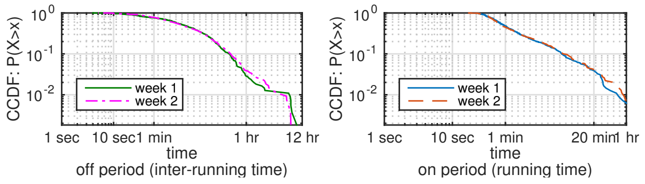


Fig. 8. The CCDF (complementary cumulative distribution functions) of off (left) and on (right) period distributions in week 1 and week 2 of user 59.

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 standby*[6] 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.

[6]Our measurement is conducted before this feature is provided.

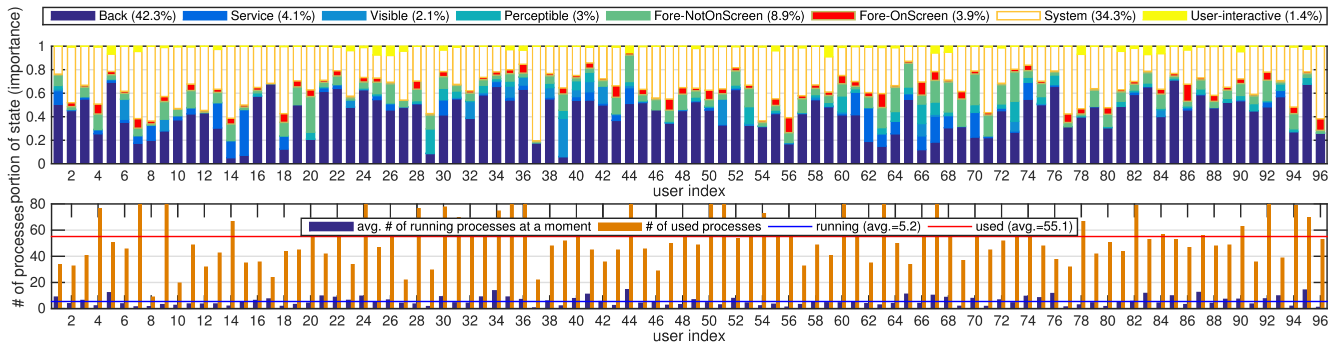


Fig. 6. The percentage of states (importance) in recorded logs (top) and the number of running processes at a moment and the number of unique processes (bottom). The numbers (-%) in the top graph indicate the average portion of each state across all participants (This figure is best viewed in color).

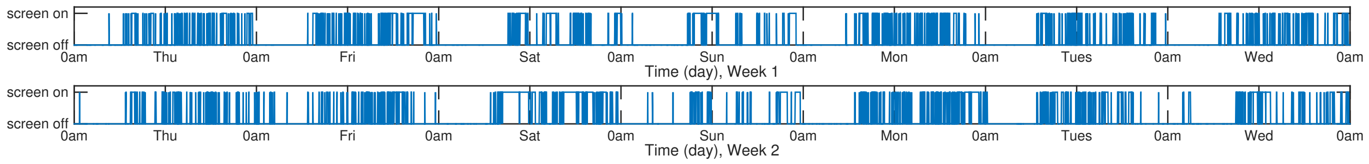


Fig. 7. Foreground activity of user 59 over two weeks (week 1 in Feb. 5-11 (top), and week 2 in Feb. 12-18 (bottom)). Regular temporal patterns are observed in weekdays and in weekends.

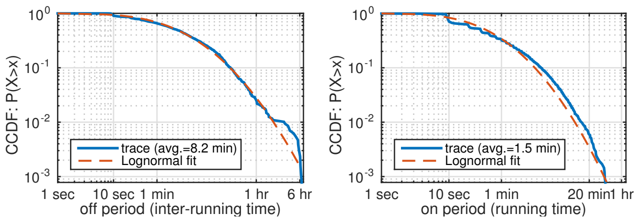


Fig. 9. The CCDF and the corresponding log-normal fittings of off (left) and on (right) periods of one randomly chosen user.

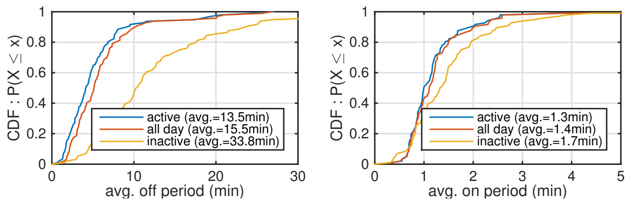


Fig. 10. The CDF of average off (left) and on (right) periods of users.

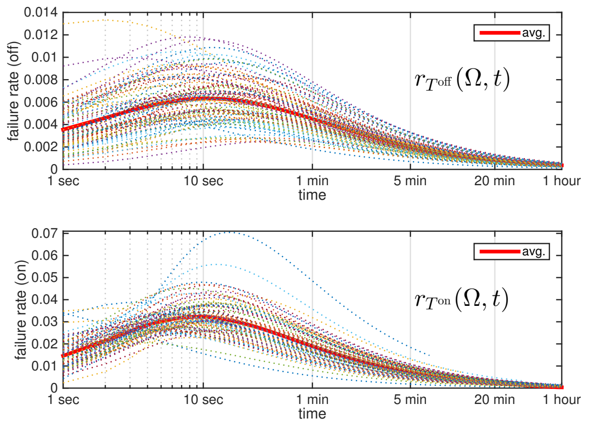


Fig. 11. Off (top) and on (bottom) failure rates of users. The length of a time slot is one second. The dotted lines are failure rates of each individual user.

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 $\tilde{r}_T(t) \triangleq \frac{f_T(t)}{1-F_T(t)}$, 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}. 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

[7]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 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.

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).

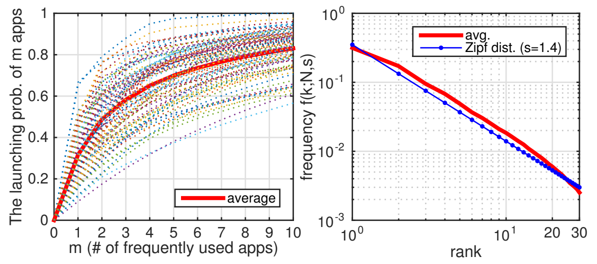


Fig. 12. The CDF of the launching probability of m most frequently used applications (left) and Zipf distribution fitting for the average launching probability (right). The frequently used applications of each user are not identical. The dotted lines are for each individual user.

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.

[8]The frequency of elements of rank k, $f(k; s, N)$ of a population of N applications is proportional to k^{-s}, where s is the exponent of the Zipfian distribution.

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

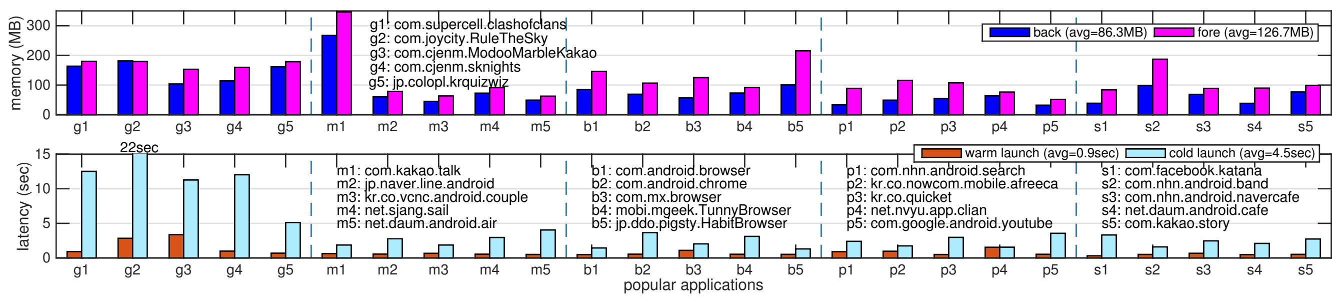


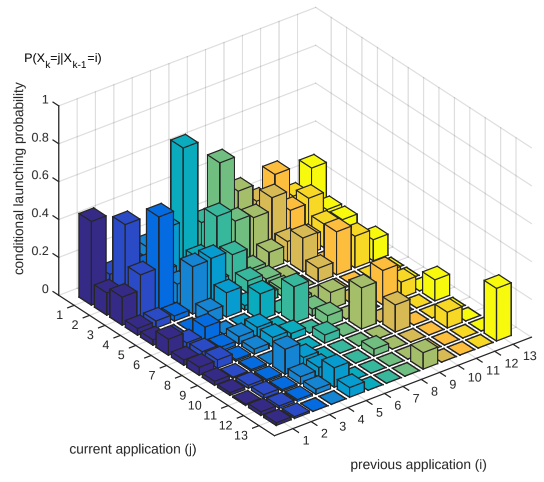
Fig. 13. Memory (top) consumption in background and foreground, and cold/warm launch latency (bottom) of popular applications in 5 categories (Game, Messaging, Browsing, Portal/Video, Social).

participants to list the applications with long startup latency and the length of perceived latency they experienced. More than 77% of participants have provided at least one application with long startup latency. The average startup latency of them is 7.3 seconds, where that of gaming applications is 9.1 seconds (and messaging: 3s, social: 3.6s, browsing: 6.3s, navigation: 6s). Thus, short lifetime, frequent freezes, and long start-up latency are still the major problems for smartphone users, even though their smartphones are almost state-of-the-art.

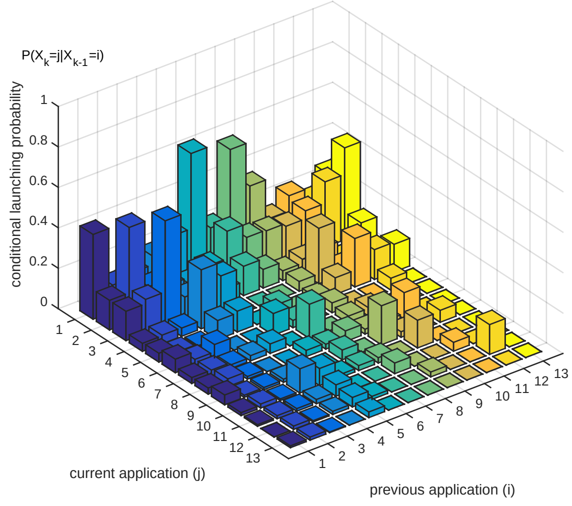
Context dependency: We also analyze app usage patterns incorporated under various contexts. Here, contexts correspond to any information that characterizes the situation of users, which enables us to predict future app/component invocations more accurately. In Fig. 10, the average inter-launching time 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 II, 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). In Fig. 14, we depict the conditional launching probability of applications (X_k) for the previously used application (X_{k-1}), for one user in each week. The application index is sorted by the launching frequency in descending order (application 1 is the most frequently launched application). We choose 13 popular applications for visibility. We note that if there is a non-zero screen-off period between two consecutive on periods (T^{on}_{k-1} and T^{on}_{k}), X_{k-1} and X_k could be the same application, and $\mathbb{P}[X_k = X_{k-1}|X_{k-1}]$ can be non-zero. We observe that the launching probability is vastly different depending on the previous applications. These patterns are also quite similar in each week. Therefore, the likelihood of launching an application at a moment depends on the previous application, and these statistics can be learned from history. We also observe context dependencies such as the duration of the previous intervals (T^{off}_{k-1}, T^{on}_{k-1}). For instance, after using a messaging app, the next inter-launching times are typically shorter than average, as the recipient of a message may respond quickly. We omit more details for brevity.

V. System Architecture

In this section, we propose our system design of CAS as depicted in Fig. 15. Our framework consists of three major



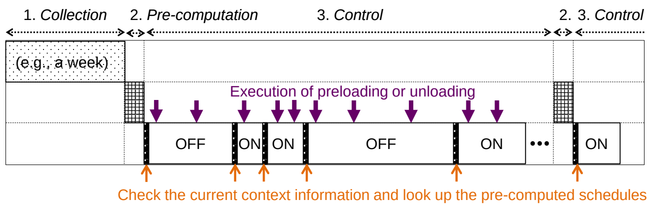
(a) Conditional launching probability (week 1).



(b) Conditional launching probability (week 2).

Fig. 14. The conditional launching probability of applications (X_k) for a previously used application (X_{k-1}) in each week of user 59.

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*. The collection phase builds personalized statistical information about application usage patterns such as described in Section IV. This data will be used in the pre-computation and


1. Collection
2. Pre-computation
3. Control
2. 3. Control
(e.g., a week)
Execution of preloading or unloading
OFF
ON
ON
OFF
ON
•••
ON
Check the current context information and look up the pre-computed schedules

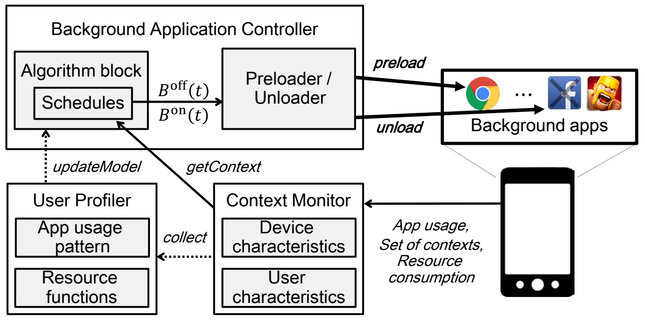


Fig. 15. The overall architecture of CAS and its operations over time.

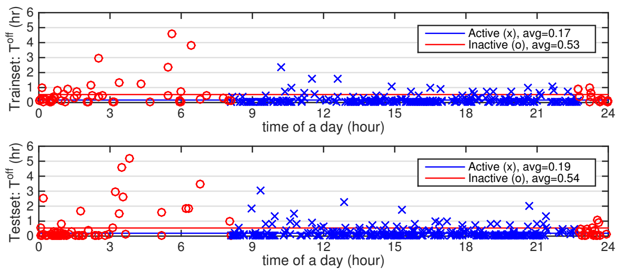


Fig. 16. A sample off-period classification by "active" (8am - 11pm) and "'inactive" (11pm - 8am) hours obtained from the first week trace (top) of one user. The classification obtained from the first week is applied to the second week (bottom) and it still shows a good match from regularity in human behaviors.

control phases. The pre-computation phase will be dicussed in more detail with the algorithm descriptions in Section VI. Here, we overview how each component works sequentially.

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 [35], [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).

To better exploit contexts, the collected context information of traces can be classified and labeled. For instance, time of a day can be roughly classified into two labels, *active* and *inactive* hours, which may show two distinct probabilistic distributions of off and on periods by the nature of human life cycles. The labels resulted from classifications of many

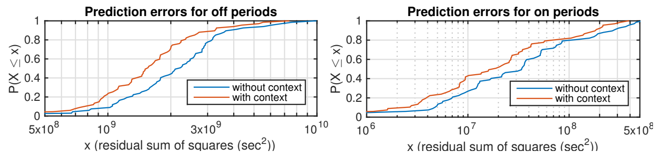


Fig. 17. CDF of prediction errors for off (left) and on (right) periods of users.

contexts will lead to a condition for prediction, where the condition is composed of a tuple of context labels such as (*time* = Active, *last app* = Facebook). People have different lifecycles and habits, so that contexts should be classified *individually*. In order to classify contexts automatically in a personalized manner without asking users a manual classification, we adopt unsupervised learning techniques. Unsupervised learning aims at classifying input data autonomously by clustering the data by their correlation (i.e., similarity). For instance, when classifying "time of a day" into k continuous time blocks, we can setup the following clustering problem which minimizes the residual sum of square errors from the clustering.[9] We solve this problem using a k-means algorithm [37].

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

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., $H_{\mathrm{A}} = [a, b]$, where a is 10:00 and b is 21:00). $S_k \in H_{\mathrm{A}}$ 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 Fig. 16. The diurnal pattern of this user is clearly identified by the classification. We find that most of the users need only 2 continuous time blocks (i.e., $k = 2$) to describe their temporal activities and show very little gain from further separation.

In Fig. 17, 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 leads to more accurate prediction. We also find that the entropy[10] of the next launching app, X_k, is substantially reduced as well, which is 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

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

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

TABLE III
SUMMARY OF MAJOR NOTATION.

Variable	Definition
$B(t) \subset \Omega$	set of background applications at time t
$\bar{B}(t) \subset \Omega$	set of empty applications at time t
$P(B(t))$	power consumption of $B(t)$
$\Delta_i P(B(t))$	$= P(B(t) \cup \{i\}) - P(B(t))$
$M(B(t))$	memory usage of $B(t)$
T_k^{off} & T_k^{on}	k-th off and on period times (random variable)
X_k	k-th launching (foreground) application (r.v.)
$r_T(\bar{B}(t), t)$	$= \frac{f_T(t)}{1-F_T(t)} \cdot \mathbb{P}[X \in \bar{B}(t)]$, failure rate of $\bar{B}(t)$
γ	trade-off parameter (power/disutility)

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.

VI. 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. We also develop an iterative algorithm that finds the optimal schedules for a given energy constraint. This constrained optimal scheduling is practically valuable to users who want the best application performance at each level of energy allowance.

A. System model

System states: We summarize major notations in Table III. 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 main control knobs.

We define the system state as an off/on period as in Fig. 5. 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}.[11] We recall that the failure rate is defined as $\tilde{r}_T(t) \triangleq \frac{f_T(t)}{1-F_T(t)}$, 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 $r_T(\bar{B}(t), t) = \tilde{r}_T(t) \cdot \mathbb{P}[X \in \bar{B}(t)]$, which quantifies the rate that one of empty applications $\bar{B}(t)$ is launched at t. Note that $r_T(\Omega, t) = \tilde{r}_T(t)$.

Power consumption and memory usage 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 usage of a set of background applications. Based on the observations made in [35], we model P as a monotone submodular function[12] of $B(t)$. Our model is reasonable since applications share hardware components, and each of them becomes more power-efficient as the utilization becomes higher. We define $\Delta_i P(B) = P(B \cup \{i\}) - P(B)$ as the marginal increase in power consumption by adding an application i in the background application set B. A memory function M is a linear additive function[13] for any $B(t)$. Note that we use average power and memory consumption for long-term optimization. For simplicity, we model that the energy consumption for preloading and unloading is minor in the long-run and that the transition delay is much shorter than one time slot. These models 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 the simulation section.

B. Problem formulation

We aim to develop a scheduling algorithm for CAS that reduces and balances energy consumption and user disutility from experiencing cold launch of applications, under a given memory budget M_{th}. Since there is a trade-off between the energy consumption and the user disutility, we adopt a parameter γ to treat both metrics as a unified measure. A user who is less sensitive to latency but is keen to extend battery lifetime will choose a smaller γ value, and vice versa.[14] 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 is formally defined below. We present the equation by the summation of two components corresponding to off and on period optimizations for better understanding.

$$\min_{\substack{B^{\text{off}}(t), \forall t \\ M(B^{\text{off}}(t)) < M_{\text{th}}}} H^{\text{off}} + \min_{\substack{B^{\text{on}}(t), \forall t \\ M(B^{\text{on}}(t)) < M_{\text{th}}}} H^{\text{on}}$$

$$H^{\text{off}} \triangleq \sum_{t=1}^{\infty} \mathbb{P}[T^{\text{off}} = t] \Big(\sum_{\tau=1}^{t} P(B^{\text{off}}(\tau)) + \gamma \cdot \mathbb{P}[X \notin B^{\text{off}}(t)] \Big),$$

[11] We omit the subscript k unless confusion arises.

[12] $P(A \cup B) \leq P(A) + P(B) - P(A \cap B)$ for $A, B \subset \Omega$, and $P(A) \leq P(B)$ for any $A \subseteq B$.

[13] For any disjoint sets $A, B \subset \Omega$, $M(A \cup B) = M(A) + M(B)$.

[14] We will discuss later in this section how γ can be automatically determined for a user who wants to limit either of energy consumption or disutility.

$$H^{\text{on}} \triangleq \sum_{t=1}^{\infty} \mathbb{P}[T^{\text{on}}=t]\Big(\sum_{\tau=1}^{t} P(B^{\text{on}}(\tau)) + \gamma^{\text{on}} \cdot \mathbb{P}[X \notin B^{\text{on}}(t)]\Big),$$

where $\gamma^{\text{on}} = \gamma \mathbb{P}[T^{\text{off}} = 0]$, $B^{\text{on}}(t) \subseteq \Omega_k$, and $\Omega_k = \Omega \setminus \{X_k\}$. We use $B^{\text{off}}(t)$ and $B^{\text{on}}(t)$ to denote the set of background applications in an off and on period, respectively.

The optimization is decomposed into off and on problems (i.e., H^{off} and H^{on}), each of which corresponds to the optimization during a screen-off or screen-on period. In the off-period optimization(H^{off}), 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. In an on period, the disutility is multiplied by the probability that the user will switch to another application without going through an off period (i.e., $\mathbb{P}[T^{\text{off}} = 0]$). Otherwise, the device will go into an off period (i.e., the user stops using the device) and there will be no disutility.

By restating energy and disutility terms in H^{off} using $\mathbb{P}[T^{\text{off}} \geq t]$, we have $H^{\text{off}} = \sum_{t=1}^{\infty} \mathbb{P}[T^{\text{off}} \geq t]\Big(P(B^{\text{off}}(t)) + \gamma \cdot r_{T^{\text{off}}}(\bar{B}^{\text{off}}(t), t)\Big)$ from the definition of the partial failue rate. Since we have assumed that the latency and energy overhead for preload and unload are minor, 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. The snapshot problem in each slot in an off period is as follows:

$$\textbf{P-off}: \min_{B^{\text{off}}(t)} \; P(B^{\text{off}}(t)) + \gamma \cdot r_{T^{\text{off}}}(\bar{B}^{\text{off}}(t), t) \tag{1}$$

$$\text{subject to} \quad M\big(B^{\text{off}}(t)\big) < M_{\text{th}}, B^{\text{off}}(t) \subseteq \Omega. \tag{2}$$

Similarly, in an on period, we have the following problem:

$$\textbf{P-on}: \min_{B^{\text{on}}(t)} P(B^{\text{on}}(t)) + \gamma^{\text{on}} \cdot r_{T^{\text{on}}}(\bar{B}^{\text{on}}(t), t)$$

$$\text{subject to} \quad M\big(B^{\text{on}}(t)\big) < M_{\text{th}}, B^{\text{on}}(t) \subseteq \Omega_k.$$

In the rest of the derivation, we focus on the **P-off** problem, as the **P-on** can be identically handled by letting $B^{\text{on}}(t) \subseteq \Omega_k$. For simplicity, 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. Thus, the objective function to minimize is rewritten as $h\big(B(t), t\big) = P(B(t)) + \gamma \cdot r_T(\bar{B}(t), t)$.

Proposition 6.1: The objective functions in **P-off** and **P-on** are submodular.

Proof: From the definition of submodularity, it is straightforward to see that the sum of a submodular function and an additive function is submodular, by subtracting the additive function in the inequality (i.e., adding an additive function to a submodular function does not break the inequality). Since $P(B(t))$ is submodular and $r_T(\bar{B}(t), t)$ is additive, the objectives in **P-off** and **P-on** are submodular. ■

C. Scheduling algorithm design

Proposition 6.1 clarifies that our problem formulation in Eq. (1) is of a constrained submodular minimization with an upper bound constraint. Note that a constrinaed submodular minimization with a lower bound cardinalty constraint has been proven to be NP-hard in [38]. Since the cardinality constraint can be generalized to rational weights and $g(S) = f(\Omega \setminus S)$ is also submodular[15] such that the upper bound constraint can be transformed to a lower bound constraint, our problem is also NP-hard. If $P(B(t))$ were additive, then this problem becomes a **0-1** knapsack problem, which can be solved using dynamic programming. For an unconstrained submodular minimization problem, Orlin et al. [39] developed an optimal polynomial-time algorithm with complexity $O(N^5L+N^6)$ where L is the time for function evaluation. The computational complexity of the optimal algorithm is already too heavy for mobile systems even without a constraint. Hence, we propose an algorithm for constrained submodular minimization with limited complexity (e.g., up to quadratic time complexity) that could result in sub-optimal performance, but in practice is often close to optimal performance. To that end, we provide necessary conditions for a policy to be optimal in Theorem 6.1. Then, we will show that our proposed policy satisfies these necessary conditions.

We denote $\pi = \big(B_\pi(t)\big)_{t=1}^{\infty}$ as a control policy. We define Π_{R} as a set of *rational control* policies as follows:

$$\Pi_{\text{R}} = \{\pi : B_\pi(t_2) \not\supseteq B_\pi(t_1), \forall t_1, t_2 \text{ s.t. } r_T(\Omega, t_2) \leq r_T(\Omega, t_1)\}.$$

The reason why it is rational is that an optimal control policy should not add more background applications in $B(t)$ when the failure rate decreases. We will show that an optimal control policy π^* satisfies $\pi^* \in \Pi_R$ in the following Theorem 6.1.

Theorem 6.1 (Necessary condition): If $B^*(t)$ is an optimal control of **P-off** in Eq. (1), then for any $i \in B^*(t)$ and $j \in \Omega \setminus B^*(t)$ such that $M(B^*(t) \cup \{j\}) \leq M_{\text{th}}$,

(i) $\Delta_i P(B^*(t) \setminus \{i\}) \leq \gamma \cdot r_T(\{i\}, t)$, and
(ii) $\Delta_j P(B^*(t)) \geq \gamma \cdot r_T(\{j\}, t)$.

Also, an optimal control policy $\pi^* = \big(B^*(t)\big)_{t=1}^{\infty}$ is in Π_{R}.

Proof: (i) Suppose that there exists $i \in B^*(t)$ such that $\Delta_i P(B^*(t) \setminus \{i\}) > \gamma \cdot r_T(\{i\}, t)$. Then $h\big(B^*(t) \setminus \{i\}, t\big) < h\big(B^*(t), t\big)$ for $h(B(t), t) = P(B(t)) + \gamma \cdot r_T(\bar{B}(t), t)$, and $B^*(t)$ is no longer an optimal control. *(ii)* can be proved in a similar manner.

Now, we will show that $\pi^* \in \Pi_{\text{R}}$ by contradiction. For any t_1, t_2 such that $r_T(\Omega, t_2) \leq r_T(\Omega, t_1)$, let $B^*(t_1)$ be an optimal control at t_1. Suppose that $B(t_2) \supsetneq B^*(t_1)$ is an optimal control at t_2. From the optimality of $B^*(t_1)$,

$$P(B(t_2)) - P(B^*(t_1)) \geq \gamma \cdot r_T(\Omega, t_1)\mathbb{P}[X \in B'],$$

where $B' = B(t_2) \setminus B^*(t_1)$. Also, since $r_T(\Omega, t_2) \leq r_T(\Omega, t_1)$, $P(B(t_2)) - P(B^*(t_1)) \geq r_T(\Omega, t_2)\mathbb{P}[X \in B']$ and $h\big(B^*(t_1), t_2\big) \leq h\big(B(t_2), t_2\big)$. Therefore, $B(t_2)$ is not an optimal control at t_2 and $\pi^* \in \Pi_{\text{R}}$. ■

We further define Π_{MR} as a set of *monotone rational control* policies as follows:

$$\Pi_{\text{MR}} = \{\pi : B_\pi(t_2) \subseteq B_\pi(t_1), \forall t_1, t_2 \text{ s.t. } r_T(\Omega, t_2) \leq r_T(\Omega, t_1)\}.$$

[15]For $S, T \in \Omega$, $g(S) + g(T) = f(\Omega \setminus S) + f(\Omega \setminus T) \geq f(\Omega \setminus (S \cap T)) + f(\Omega \setminus (S \cup T)) = g(S \cap T) + g(S \cup T)$.

By the definition, $\Pi_{\text{MR}} \subseteq \Pi_{\text{R}}$. A monotone rational control policy tends to minimize the number of control actions (i.e., preloads/unloads), which in turn reduces the control overhead as shown in Proposition 6.2.

Proposition 6.2 (Control overhead): For a monotone rational control policy $\pi \in \Pi_{\text{MR}}$, if $r_T(\Omega, t)$ is unimodal with t, both the numbers of control actions (i.e., preloads and unloads) are less than or equal to N.

Proof: Suppose that $r_T(\Omega, t)$ is unimodal with t and its maximum is at τ. For $t \leq \tau$, $r(\Omega, t)$ is non-decreasing and $B_\pi(t_1) \subset B_\pi(t_2)$ for any $t_1, t_2 \in [1, \tau]$ such that $t_1 < t_2$. The number of preloads in $[1, \tau]$ is $\sum_{t=1}^{\tau} |B_\pi(t) \setminus B_\pi(t-1)| = \sum_{t=1}^{\tau} \big(|B_\pi(t)| - |B_\pi(t-1)|\big)$. Thus, the number of preloads for $t \in [1, \tau]$ is less than or equal to N since $|B_\pi(t)| \leq N, \forall t$ and there is no unloading. For $t > \tau$, $r(\Omega, t)$ is non-increasing and and $B(t_1) \supset B(t_2)$ for any $t_1, t_2 \in [1, \tau]$ such that $t_1 < t_2$. The number of unloads in (τ, ∞) is $\sum_{t=\tau+1}^{\infty} |B_\pi(t-1) \setminus B_\pi(t)| = \sum_{t=\tau+1}^{\infty} \big(|B_\pi(t-1)| - |B_\pi(t)|\big)$, and it is less than or equal to N, and there is no preloading. ■

CAS scheduling algorithm: We propose a greedy-based algorithm that makes locally optimal choices in finding a set of background applications, and thus satisfies the necessary conditions in Theorem 6.1. 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. Our scheduling algorithm also incurs small control overhead from Proposition 6.2, as we will show that the obtained policy is a monotone rational policy.

CAS-Scheduler(γ)

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

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.

It is easy to see that the obtained policy from our scheduling algorithm is a monotone rational policy. Also, the obtained control policy satisfies the necessary conditions for optimality in Theorem 6.1, since it makes locally optimal choices in line 3 of step (A) and stops increasing the background application set when there is no improvement in the objective function in line 2 and 3 of step (B). We also 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 (e.g., once in a day or even less frequently). At the run time, the predetermined schedule is just being executed. In our CAS architecture in Section V, the policy wakes the device up only when there is an action to apply (either of preload or unload), and does nothing otherwise, to minimize energy overhead.

Using contextual information: In our framework, we can use more elaborate values of partial failure rates of applications (i.e., off/on period distributions and next application probabilities) using surrounding contexts such as the previously used application (X_{k-1}), time of a day (Z_k), location (L_k), previous time durations (T_{k-1}^{off} and T_{k-1}^{on}). Each context is monitored and recognized at the beginning of each off/on period. If context set C is detected, that period will use the conditional distribution $T_k|C$ and conditional probability of $X_k|C$ where these conditional values had been trained under the context set C. We will show the performance benefit from exploiting contextual information in Section VII.

D. Bisection method for an energy or disutility constrained optimization

Although it is possible for a user to jointly optimize energy and application launching latency through our framework, it is often more straightforward to optimize the latency performance given an energy constraint that coincides with the user's charging pattern. For this, we consider the following energy constrained problem, where our original problem ($\min H^{\text{off}}$)[16] can be viewed as the Lagrange relaxation problem of this problem.[17]

$$\min_{\substack{B^{\text{off}}(t), \forall t \\ M(B^{\text{off}}(t)) < M_{\text{th}}}} \sum_{t=1}^{\infty} \mathbb{P}[T^{\text{off}} = t] \mathbb{P}[X \notin B^{\text{off}}(t)], \tag{3}$$

$$\text{subject to } \sum_{t=1}^{\infty} \mathbb{P}[T^{\text{off}} \geq t] P(B^{\text{off}}(t)) \leq V, \tag{4}$$

where V is the average energy constraint for an off period. The optimal objective of the Lagrange relaxation problem will be no smaller than the optimal objective of the problem (3). This

[16]We focus on the off problem for simplicity.

[17]The trade-off parameter γ is a reciprocal of the Lagrange multiplier λ.

gap can become smaller as we have shorter time slots and more fine-grained control of background applications. To that end, we approximate the solution of the energy constrained problem with the original weighted sum minimization problem. In particular, we devise an iterative algorithm to find the trade-off parameter γ, under which our scheduling algorithm meets the given energy constraint. As the trade-off parameter γ increases, more applications will be scheduled in background in a monotone rational control policy (as well as in our scheduling algorithm from line 2 of Step (B)), so that the energy consumption increases and disutility decreases. In other words, for any monotone rational control policy, the energy consumption is non-decreasing and disutility is non-increasing in γ.

To find the trade-off paramter γ, under which the obtained policy satisfies the given energy constraint, we use a bisection method (similar to [40]), which is reliable if the initial interval $[\gamma_1, \gamma_2]$ is chosen appropriately. Note that since both the objective function and the constraint are neither continuous nor differentiable, we cannot apply first-order or second-order iteration algorithms (e.g., gradient descent or Newton's method) that are faster than the bisection method in specific conditions. One can apply a quasi-Newton method, but the convergence is guaranteed under specific conditions including Lipschitz continuity.

From the non-decreasing property of the energy consumption with respect to γ in the weighted sum minimization problem, there exists $\gamma \in [\gamma_1, \gamma_2]$ that satisfies the energy constraint with the smallest error in the interval,[18] if γ_1 yields less energy than V and γ_2 has higher energy than V, where V is the average energy budget in an off period to satisfy the given lifetime constraint. In each iteration, the method computes the energy of the middle point, $\gamma_m = \frac{\gamma_1+\gamma_2}{2}$, and chooses the half interval (either $[\gamma_1, \gamma_m]$ or $[\gamma_m, \gamma_2]$), in which the solution exists. The formal iteration algorithm is as follows.

Bisection method for a given energy constraint V

```
input: $V, \epsilon$, $\gamma_1 = 0$, sufficiently large $\gamma_2$
output: $\gamma_m, B(1), \cdots, B(t_{\max})$
1: while $\gamma_2 - \gamma_1 > \epsilon$,
2:     $\gamma_m = (\gamma_1 + \gamma_2)/2$.
3:     $(B(1), \cdots, B(t_{\max})) \leftarrow$ CAS-Scheduler($\gamma_m$).
4:     $E = \sum_{t=1}^{t_{\max}} \mathbb{P}[T^{\text{off}} \geq t] P(B(t))$.
5:     if $E \geq V$, $\gamma_2 = \gamma_m$; else $\gamma_1 = \gamma_m$.
```

Since the interval becomes half in each iteration, the number of iterations to converge is $\lceil \log_2(\frac{\gamma_2-\gamma_1}{\epsilon}) \rceil$. In other words, the rate of convergence of the bisection method is 1/2, where the rate of convergence is $\lim_{k\to\infty} \frac{\gamma_{k+1}-\gamma^*}{\gamma_k-\gamma^*}$, where γ_k is the value at k-th iteration, and γ^* is the weight such that the energy term is equal to the energy constraint, i.e., $E = V$. Note that the iteration algorithm can be easily generalized to consider both off and on periods, by considering the time-averaged energy consumption. Also, the iteration algorithm can be similarly applied to the constraint on disutility from application launch latency. We will see the convergence of our iteration algorithm in Section VII.

[18]Note that a mixed policy (that takes deterministic policies with some probabilities) can make the energy consumption continuous, so that the solution exists within this interval by the intermediate value theorem. We assume that the time slots are sufficiently short such that the error can be smaller than the given tolerance (ϵ).

TABLE IV
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

VII. Trace-driven simulation

A. 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 [35] 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 [27] in Android 6.0, BFC and HUSH proposed in [2] with CAS. The contextual information we used is summarized in Table IV.[19] 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 [2]. The memory threshold for CAS is set to be 30% of the total memory size for each user leading to 840MB on average.[20]

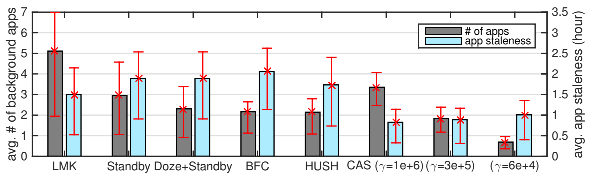
B. Key Results

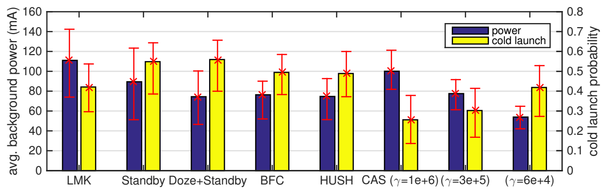
We depict the performance of different scheduling algorithms in Fig. 18, and summarize results as follows.

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

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

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


(a) Number of background applications and staleness.


(b) Background power and cold launch probability.

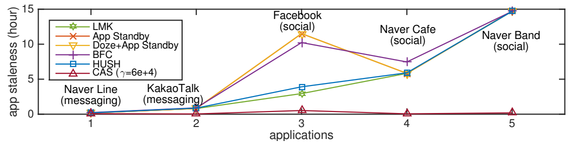

(c) Staleness of social and messaging apps on one user.

Fig. 18. Comparison of scheduling algorithms. The error bars indicate 25th and 75th percentiles.

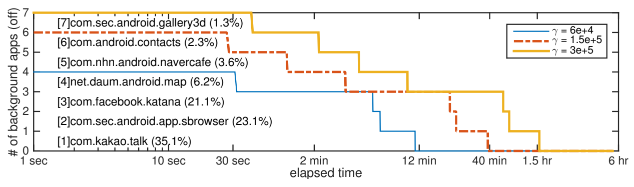


Fig. 19. Background application schedules of CAS for one user in off periods. The x-axis is in log scale. We list the application names of the ordered sequence in the graph and the numbers (-%) indicate the launching probabilities. 1: Messaging, 2: Browsing, 3, 5: Social, 4: Navigation, 6: Contacts, 7: Utility.

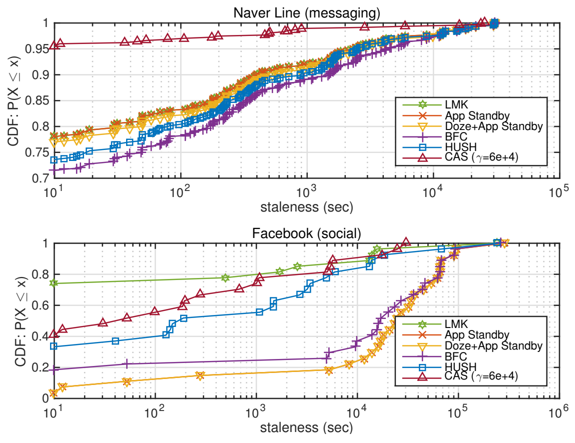


Fig. 20. CDF of staleness in two applications, Naver Line and Facebook, on one user.

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.[21]

Android 6.0, BFC and HUSH [2]: The new feature, *App standby* [27], in Android 6.0 unloads applications that have no foreground activity for more than 3 days. The portion of 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 Fig. 18(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 [27] 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 Fig. 19. Also, according to our finding that the launching probability decreases as the elapsed time passes by in each off/on period (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, γ = 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 γ = 6e+4 in Fig. 18. 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.

[21]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.

Staleness: We define *app staleness* as the average elapsed time since the last background or foreground activity of an application as in [2]. 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 Fig. 18(a). Unlike other algorithms, CAS preloads background applications and reduces the average staleness by 33% and 41% over LMK for γ = 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 Fig. 18(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. We also depict the CDF of staleness for two applications, Naver Line and Facebook, on the same user, in Fig. 20. In the Naver Line application, the staleness from CAS is stochastically less than that of any other algorithms. Other applications show similar performance characteristics. In Facebook, LMK has a higher probability for small staleness (e.g., less than 1000 sec) than CAS, but the average staleness of CAS is still smaller than that of LMK. This is because CAS preloads Facebook once in an off or on period most of the time so as to prevent the extremely long staleness.

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.

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 (see Fig. 13) and summarize them in Table V. We include a simulation of the ideal yet infeasible scheduler, *Oracle*, that exactly knows when and which application the user will use next. Note that the upper bound of battery lifetime simulated from *Oracle* is 21.9 hours. For CAS, we apply the bisection method in Section VI-D to achieve the same disutility or energy as LMK, and the same energy as HUSH. The average battery lifetime of a device is extended to 16.3 hours in CAS from 12.2 hours observed under LMK, with the same disutility (i.e., the expected start-up latency). When the lifetime of CAS is equalized to that of LMK, the expected start-up latency is reduced from 2.45sec in LMK to 1.85sec in CAS. When the lifetime of CAS gets equalized to that of HUSH, CAS can reduce the expected start-up latency from 2.66sec to 2.24sec.

TABLE V
COMPARISON OF SCHEDULING ALGORITHMS.

Scheduler	Power$^\sharp$	Cold launch	Memory$^\sharp$	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 [2]	76.4mA	49.4%	93MB	14.4 hr	2.67sec
HUSH [2]	74.7mA	48.9%	91MB	14.5 hr	2.66sec
CAS (same energy as LMK)	111mA	26.3%	138MB	12.2 hr	1.85sec
CAS (same energy as HUSH)	74.7mA	35.1%	95MB	14.5 hr	2.13sec
CAS (same disutility as LMK)	54.2mA	43%	58MB	16.3 hr	2.45sec
Oracle	10mA	0%	0MB	21.9 hr	0.9sec

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

VIII. 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. The implementation of context monitor specifically for CAS focuses on collecting application usage history and time of the day information that incurs minimal energy overhead, 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>*[22] 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)

[22]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 VI
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[23] such as the 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 VI. 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.

[23]This interception is similarly implemented as [2].

IX. 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.

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 [14]. We can also use crowdsourced statistics[24] of applications to minimize the need for user inputs. As more applications are whitelisted, the possibility of energy saving is reduced as well. Because whitelisted applications will work as intended by developers without considering other applications and application usage behaviors, this may incur energy inefficiency of the entire system.

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.

X. CONCLUDING REMARKS

Our scheduling framework, CAS, is the first work that considers both preloading and unloading dimensions in application scheduling. Through CAS, we have shown in this paper that mobile systems can achieve much higher efficiency

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

in resource management than conventional systems by understanding human behaviors on application usage and related contextual information. Trace-driven simulations demonstrate that CAS outperforms LMK, HUSH, and Android 6.0. We also implement CAS on Android and validate the performance through experiments. We underscore that the significant gain of CAS is from regularity and context-dependency of human behaviors.

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.

References

[1] J. Lee, K. Lee, E. Jeong, J. Jo, and N. B. Shroff, "Context-aware Application Control in Mobile Systems: What Will Users Do and Not Do Next?" in *Proc. of ACM UbiComp*, 2016.

[2] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone background activities in the wild: Origin, energy drain, and optimization," in *Proc. of ACM MobiCom*, 2015.

[3] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone energy drain in the wild: Analysis and implications," in *Proc. of ACM Sigmetrics*, 2015.

[4] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen, "Revisiting network energy efficiency of mobile apps: Performance in the wild," in *Proc. of ACM Conference on Internet Measurement Conference (IMC)*, 2015.

[5] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proc. of ACM EuroSys*, 2012, pp. 29–42.

[6] N. Ding, D. Wagner, X. Chen, Y. C. Hu, and A. Rice, "Characterizing and modeling the impact of wireless signal strength on smartphone battery drain," in *Proc. of the ACM SIGMETRICS*, 2013.

[7] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4g lte networks," in *Proc. of ACM MobiSys*, 2012.

[8] Android Developers, "Managing Your App's Memory," 2015, https://developer.android.com/training/articles/memory.html.

[9] Apple Developer, "App programming guide for ios: The app life cycle," 2015, https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html.

[10] J. H. Kim, J. Sung, S. Y. Hwang, and H.-J. Suh, "A novel android memory management policy focused on periodic habits of a user," in *Ubiquitous Computing Application and Wireless Sensor*, 2015, pp. 143–149.

[11] A. J. Oliner, A. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica, "Collaborative energy debugging for mobile devices," in *Proc. of USENIX Workshop on Hot Topics in System Dependability (HotDep)*, 2012.

[12] A. Pathak, Y. C. Hu, and M. Zhang, "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)*, 2011.

[13] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proc. of ACM MobiSys*, 2012.

[14] I. Singh, S. V. Krishnamurthy, H. V. Madhyastha, and I. Neamtiu, "Zapdroid: managing infrequently used applications on smartphones," in *Proc. of ACM UbiComp*, 2015, pp. 1185–1196.

[15] H. Verkasalo, "Contextual patterns in mobile service usage," *Personal and Ubiquitous Computing*, vol. 13, no. 5, pp. 331–342, 2009.

[16] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "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)*, 2011.

[17] C. Shin, J.-H. Hong, and A. K. Dey, "Understanding and prediction of mobile application usage for smart phones," in *Proc. of ACM UbiComp*, 2012.

[18] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "LiveLab: measuring wireless networks and smartphone users in the field," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, pp. 15–20, 2011.

[19] A. Rahmati, C. Shepard, C. Tossell, L. Zhong, and P. Kortum, "Practical context awareness: Measuring and utilizing the context dependency of mobile usage," *IEEE Transactions on Mobile Computing*, 2014.

[20] T. M. T. Do, J. Blom, and D. Gatica-Perez, "Smartphone usage in the wild: a large-scale analysis of applications and context," in *Proc. of ACM conference on multimodal interfaces*, 2011.

[21] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proc. of ACM MobiSys*, 2010, pp. 179–194.

[22] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proc. of ACM MobiSys*, 2012.

[23] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin, "Practical prediction and prefetch for faster access to applications on mobile phones," in *Proc. of ACM UbiComp*, 2013.

[24] N. Natarajan, D. Shin, and I. S. Dhillon, "Which app will you use next?: Collaborative filtering with interactional context," in *Proc. of ACM Recommender systems (RecSys)*, 2013.

[25] R. Baeza-Yates, D. Jiang, F. Silvestri, and B. Harrison, "Predicting the next app that you are going to use," in *Proc. of ACM Web Search and Data Mining*, 2015.

[26] Y. Wang, X. Liu, D. Chu, and Y. Liu, "Earlybird: Mobile prefetching of social network feeds via content preference mining and usage pattern analysis," in *Proc. of ACM MobiHoc*, 2015.

[27] Android Developers, "Optimizaing for Doze and App Standby," 2016, https://developer.android.com/training/monitoring-device-state/doze-standby.html.

[28] ——, "ActivityManager.RunningAppProcessInfo," 2015, http://developer.android.com/reference/android/app/ActivityManager.RunningAppProcessInfo.html.

[29] J. Lee, K. Lee, J. Jo, E. Jeong, and N. B. Shroff, "CAS Survey Questions and Results," School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Tech. Rep. 1, 2015, http://msn.unist.ac.kr/CAS_survey.pdf.

[30] W. T. Eadie, M. G. Roos, and F. E. James, *Statistical Methods in Experimental Physics*. Elsevier Science and Technology, 1971.

[31] K. P. Burnham and D. R. Anderson, "Multimodel inference: Understanding aic and bic in model selection," *Sociological Methods and Research*, vol. 33, no. 2, pp. 261–304, 2004.

[32] J. Jeong, Y. Yi, J.-w. Cho, D. Y. Eun, and S. Chong, "Wi-fi sensing: Should mobiles sleep longer as they age?" in *IEEE INFOCOM*, 2013.

[33] Android Developers, "Android logcat," 2015, http://developer.android.com/tools/help/logcat.html.

[34] Advanced Task Killer, 2015, https://play.google.com/store/apps/details?id=com.rechild.advancedtaskkiller.

[35] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "AppScope: Application energy metering framework for android smartphone using kernel activity monitoring." in *Proc. of USENIX ATC*, 2012.

[36] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha, "DevScope: a nonintrusive and online power analysis tool for smartphone hardware components," in *Proc. of ACM Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2012.

[37] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[38] Z. Svitkina and L. Fleischer, "Submodular approximation: Sampling-based algorithms and lower bounds," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1715–1737, 2011.

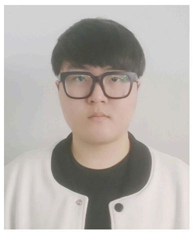
[39] J. B. Orlin, "A faster strongly polynomial time algorithm for submodular function minimization," *Mathematical Programming*, vol. 118, no. 2, pp. 237–251, 2009.

[40] R. Cendrillon, W. Yu, M. Moonen, J. Verlinden, and T. Bostoen, "Optimal multiuser spectrum balancing for digital subscriber lines," *IEEE Transactions on Communications*, vol. 54, no. 5, pp. 922–933, 2006.

Joohyun Lee (S'11-M'14) received his B.S. and integrated M.S./Ph.D degree in the Department of Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2008 and 2014, respectively. He is currently a postdoctoral researcher in the Department of Electrical and Computer Engineering at the Ohio State University. He received IEEE William R. Bennett Prize Paper Award in 2016, given to the best original paper published in IEEE/ACM Transactions on Networking in the previous three calendar years. His research interests are in the areas of context-aware networking and computing, mobility-driven cellular traffic offloading, energy-efficient mobile networking, protocol design and analysis for delay-tolerant networks, and network economics and pricing.

Kyunghan Lee (S'07-M'10) is an associate professor in the school of electrical and computer engineering at UNIST (Ulsan National Institute of Science and Technology), Ulsan, South Korea. His research interests include low latency networking, low-power mobile computing, and mobility-aware networking. He received two IEEE ComSoc William R. Bennett Prize Paper Awards in 2013 and 2016, given to the best original paper published in IEEE/ACM Transactions on Networking in the previous three calendar years. He received his B.S., M.S., and Ph.D. degrees in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2002, 2004, and 2009, respectively.

Euijin Jeong received his B.S. in Computer Science and Engineering from Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea, in 2016. He is currently studying for a master degree at Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea. His research interests include the areas of context-aware networking and computing, mobile usage prediction, and mobile energy saving.



Jaemin Jo received his B.S. in Computer Science and Engineering from Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea, in 2016. He is currently studying for a master degree at Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea. His research interests include the areas of context-aware networking, architecture design, and mobile energy saving.

Ness B. Shroff (S'91-M'93-SM'01-F'07) received his Ph.D. degree in Electrical Engineering from Columbia University in 1994. He joined Purdue university as an Assistant Professor in the school of ECE, where he became a Full Professor of ECE in 2003 and director of CWSA in 2004, a university-wide center on wireless systems and applications. In July 2007, he joined The Ohio State University, where he holds the Ohio Eminent Scholar endowed chair in Networking and Communications, in the departments of ECE and CSE. He holds or has held visiting (chaired) professor positions at Tsinghua University, Beijing, China, Shanghai Jiaotong University, Shanghai, China, and the Indian Institute of Technology, Bombay, India. Dr. Shroff is currently an editor at large of IEEE/ACM Trans. on Networking, senior editor of IEEE Transactions on Control of Networked Systems, and technical editor for the IEEE Network Magazine. He has received numerous best paper awards for his research and listed Thomson Reuters Book on The World's Most Influential Scientific Minds as well as noted as a highly cited researcher by Thomson Reuters. He also received the IEEE INFOCOM achievement award for seminal contributions to scheduling and resource allocation in wireless networks.