Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization

Xiaomeng Chen[†^] Abhilash Jindal[†^] Ning Ding[†^] Y. Charlie Hu[†^] Maruti Gupta[⋆]
Rath Vannithamby[⋆]
[†]Purdue University [^]Mobile Enerlytics [⋆]Intel Corporation

ABSTRACT

As new iterations of more powerful and better connected smartphones emerge, their limited battery life remains a leading factor adversely affecting the mobile experience of millions of smartphone users. While it is well-known that many apps can drain battery even while running in background, there has not been any study that quantifies the extent and severity of such background energy drain for users in the wild. To extend battery life, various new features are being incorporated within the phone, one of them being preventing applications from running in background, *i.e.,* when the screen is off, but their impact is largely unknown.

This paper makes several contributions. First, we present a large-scale measurement study that performs an in-depth analysis of the activities of various apps running in background on thousands of phones in the wild. Second, we quantify the amount of battery drain by all such background activities and possible energy saving. Third, we develop a metric to measure the usefulness of background activities that is personalized to each user. Finally, we present a system called HUSH (screen-off optimizer) that monitors the metric online and automatically identifies and suppresses background activities during screen-off periods that are not useful to the user experience. In doing so, our proposed HUSH saves screen-off energy of smartphones by 15.7% on average while incurring minimal impact on the user experience with the apps.

Categories and Subject Descriptors

C.4 [**Computer System Organization**]: Performance of Systems—*Modeling techniques*; D.2.8 [**Metrics**]: Performance measures

General Terms

Experimentation, Measurement, Performance

Keywords

Smartphones, background activities, screen-off energy drain

1. INTRODUCTION

It is well known that many apps on smartphones wake up periodically to run when users are not actively interacting with them. This happens predominantly during screen-off periods but can also happen when other apps are running in the foreground during screen-on periods. The reasons for such periodic wakeups are severalfold: (1) to perform a refresh of the app state, *e.g.,* in apps that provide updates for news, financial stocks, weather or twitter feed updates, (2) to sync with cloud services and get status updates or notifications, *e.g.,* in social networking apps such as Facebook, WeChat, and Google+, (3) to support non-touch based user interactions such as in audio apps Pandora and Spotify which periodically download files to play songs, or apps used for location tracking or navigation which allow a user to listen to directions without screen interactions. Additionally, many system services may run in the background due to either direct invocations by apps (*e.g.,* MediaPlaybackService) or the registration and callback mechanism where an app during initiation registers for future events monitored by the system services (*e.g.,* LocationManagerService).

More generally, many mobile apps are designed to run in background to enable a model of always-on connectivity and to provide fast response time. This means that once installed and initiated by the user, apps can register themselves with the services provided by the OS framework for background activities, regardless of the user's actual usage of the app. This is true of both iOS and Android OS [2, 7]. This leads one to the conjecture that the apps running in background and the system services they invoke may be consuming a significant amount of energy. Despite such general perception, there has not been any quantitative measurement of the background activities and energy drain by apps running on user phones in the wild. To quantify such app background activities and energy drain on real users' phones, we conducted a large-scale measurement of 2000 Samsung Galaxy phones in the wild. Our analysis shows that background activities of apps and services are prevalent and on average 28.9% of the daily energy drain is due to apps and services running during screen-off intervals.

Given the perception of the effects of app background activities on the phone's battery, it is not surprising that smartphones now come with features to turn off background activities, *e.g.,* iOS provides a feature called "Disable Background Refresh" [6] which basically prevents the apps from running in the background altogether. Android has a similar feature called "Restrict Background Data" [1] which prevents the user from running up charges on their mobile data usage due to background data activities by switching background data refresh to using WiFi only. Such blanket solutions disable background activities of all apps and hence do not distinguish background activities of apps that are potentially more useful to the user from the rest, which can adversely affect user experience. Some apps provide the user with the option of disabling background activities. However, it is very cumbersome for a user to manually change the settings for every installed app on the phone.

MobiCom'15, September 07–11, 2015, Paris, France.

ACM 978-1-4503-3619-2/15/09 ...$15.00 .
DOI: http://dx.doi.org/10.1145/2789168.2790107 .

In this paper, we explore effective ways of optimizing background activities of apps and services. Our exploration is motivated by two hypotheses: (1) background activities of apps are meant to improve user app experience but they are only useful if the user interacts with those apps in foreground sometime during the next screen-on interval; (2) the usefulness of background activities of an app is likely to be user-dependent and thus their occurrences should be personalized.

First, we propose a metric to measure this usefulness of app background activities called Background to Foreground Correlation (BFC). Second, we experimentally confirm the hypotheses using our 2000-user trace. The confirmation suggests the level of background activities of an app should be *personalized* to individual users. We then design a screen-off energy optimizer on Android called HUSH that monitors the BFC of all apps on a phone online and automatically identifies and suppresses app background activities during screen-off intervals that are not useful to the user experience. In doing so, HUSH saves screen-off energy of smartphones by 15.7% on average with minimal impact on the user experience with the apps.

2. RELATED WORK

There have been recently a lot of interests in measurement studies of smartphone app traffic and their impact on power consumption. In [13], Falaki *et al.* characterized smartphone traffic based on traffic traces collected from 43 users and showed how power consumption could be reduced. In [16, 14], Huang *et al.* studied 3G and LTE network performance and described the impact of the RRC radio power states on radio energy consumption. In [26], Xu *et al.* studied the smartphone usage patterns via network measurement from a tier-1 cellular network provider. AppInsight [23] monitors the performance of mobile apps in the wild by instrumenting app binary. In [25], Sommers and Barford studied the WiFi and cellular performance using the Speedtest.net data. None of the work above, however, provided a thorough analysis of how energy is consumed on the smartphones in the wild.

In [15], Huang *et al.* collected traffic from 20 users and studied screen-off radio energy consumption due to cellular network and proposed to provide savings by shifting this traffic load over WiFi instead, on an opportunistic basis. In [26], Xu *et al.* studied the energy drain of email sync operations of two email apps when the phone is in standby mode (*i.e.,* during screen-off) using a power-meter and proposed ways to optimize the email sync operations.

In summary, while there have been many studies recently that characterize the traffic seen on smartphones, only a few have looked at the impact of background traffic activities on the power consumption, *e.g.,* on network components such as WiFi and cellular radios, using small-scale measurement or in the lab setting.

Most recently, Chen *et al.* performed the first large-scale measurement study of energy drain on Android phones in the wild. Their focus is on a holistic analysis that comprehensively breaks down the total daily energy drain into different phone components as well as apps and services running on the phones and further draws implications to the designers of different components of smartphone and app developers.

The first part of this paper also presents a large-scale measurement study of Android phones in the wild, but the focus is on a detailed analysis of the activity, origin, and energy drain of background apps and services. Second, we introduce a new metric that characterizes the usefulness of background activities in a way that is customized to each app and each user. Finally, we present a novel technique called HUSH with implementation on Android phones that reduces the energy consumption due to background activities and show its energy impact on actual phones.

Table 1: Trace statistics.

Devices ($\geq$ 7 days trace)	2000
Aggregate trace duration	55759 days
Median trace duration	27.9 days
Countries of origin	61
Mobile operators	191
Unique phone types	2 (Galaxy S3 and S4)
Rate of mobile RSSI reading	when signal changes, effective: 1/min
Rate of network usage reading	screen-on: every 1 second, screen-off: every 5 seconds

3. CHARACTERIZING BACKGROUND ACTIVITIES IN THE WILD

In this section, we study the background activities on the 2000 phones in the wild.

3.1 Trace Collection

To perform the measurement study, we designed a free Android app called eStar [4] that when installed on a user's phone, performs periodic logging of the usage of various phone components by apps and system services. As discussed in Section 4, the logged information is also sufficient to drive a carefully designed power model to estimate the energy drain of the apps and services and various phone components in each logging interval. Since the app does not require any modifications to the Android framework or the kernel, its logging can be performed on smartphones in the wild. All the information collected (summarized below) are anonymized before uploaded to our server.

In principle, the shorter the logging interval, the more fine-grained the collected information, which leads to better accuracy in estimating energy drain, but also the higher the logging overhead. In designing the app, we carefully chose logging intervals to strike a balance between these two objectives.

Coarse-grained logging. Coarse-grained logging happens every 5 minutes, where the app logs app-wise CPU usage (from */proc/[pid]/stat*), and the per-core CPU usage (from */proc/stat*) and the time duration spent in different frequencies (from */sys/devices/system/cpu/cpu[id]/*).

Fine-grained logging. Fine-grained logging happens every 5 seconds during screen-off when CPU is on and every 1 second during screen-on, where the app logs the network usage (in bytes) of all apps that had data transfer during the interval, by reading */proc/uid_stat/[uid]/*.

On-demand event logging. Finally, dynamic events are logged on demand. These include WiFi, mobile data, and screen being switched on and off, WiFi being associated and scanning, WiFi and cellular signal strength change, battery level change (1% granularity), and the time each app starts and stops.

Trace collection. We released the eStar app in Google Play on September 1, 2014 which was downloaded by thousands of users worldwide [1]. The trace we used in this study contains logs from 2000 Galaxy S3 and S4 devices. Each user trace ranges from 7 to 40 days in length, with an average of 27.9 days (median 30 days). The detailed characteristics of the trace are shown in Table 1.

Logging overhead. The logging was shown in [11] to incur very low overhead: the average overhead per day across the devices in the trace were 2.4% of the total CPU time, 0.3% of the total network bytes, and 0.6% of the total energy drain.

[1]eStar received exemption from the full requirement of 45 CFR 46 or 21 CFR 56 by the IRB of our organization.

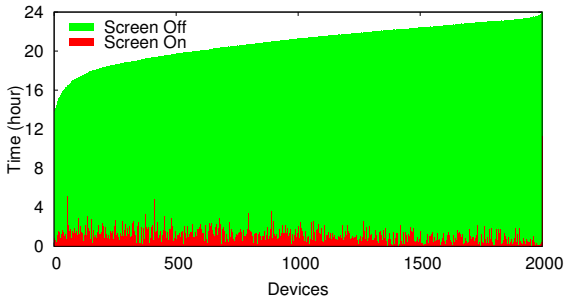


Figure 1: Daily screen-on vs screen-off time across users.

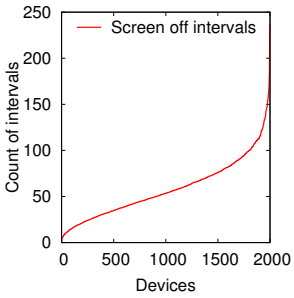


Figure 2: Average number of screen-on (and hence screen-off) intervals per day across users.

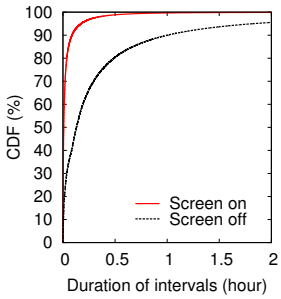


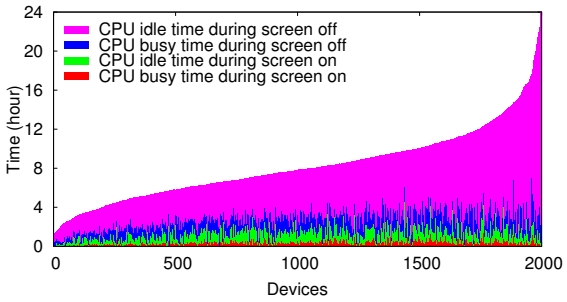
Figure 3: CDF of screen-on and screen-off interval durations across users for all days.

3.2 Screen-off Intervals

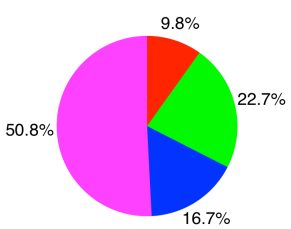
We start with an overview of the smartphone foreground and background usage across the 2000 devices. Figure 1 shows the distribution of the average daily total time spent in screen-off and screen-on intervals across the 2000 devices sorted by the total time. The reason the screen-off and screen-on time do not add up to 24 hours is that a phone can be turned off. On average, the users spend around 2.56 hours per day actively interacting with their phones; the 10th percentile, and 90th percentile screen-on time are 0.84 and 4.62 hours, respectively.

Next, we study the frequency and duration of screen-on and screen-off intervals. Figure 2 shows other than the 9.3% users who on average turn their phones on over 100 times a day, the usage by the rest of the users is spread almost uniformly between 2 to 100 times a day. Overall, the average, 10th percentile, and 90th percentile numbers of screen-on/screen-off intervals per day are 58.3, 21.4, 99.0, respectively. The last user (the rightmost) on average turned the device on 237 times a day for a total average daily screen-on time of 4.2 hours, to mainly check two apps – Whatsapp and Instagram.

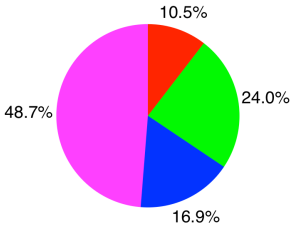
The roughly similar total screen-on time (Figure 1) but diverse numbers of screen-on intervals suggest the screen-on and screen-off intervals (Figure 2) differ significantly in duration. Figure 3 plots the CDF of the duration of individual screen-on/off intervals for all users across all days, truncated at 2 hours. We observe that as expected, (1) the duration of screen-on/off intervals differ significantly, and (2) the screen-off intervals tend to last much longer than screen-on intervals. The average, 10th percentile, and 90th percentile durations are 2.9 minutes, 4.9 seconds, and 5.2 minutes for screen-on intervals, and 26.8 minutes, 18.0 seconds, and 60.0 minutes for screen-off intervals, respectively.



(a) Percentage CPU time breakdown of all devices.



(b) Average of percentage breakdown of CPU time across all devices.



(c) After removing USB charging intervals and CPU lingering time.

Figure 4: Distribution of daily CPU busy and idle time during screen-on and screen-off periods.

3.3 Background Activities in Screen-off

A longer screen-off interval does not necessarily imply more background activities and more energy drain. To understand background activities and energy drain, we first measure the CPU time spent to perform background activities. To this end, we break down the total CPU time of a device, *e.g.,* in a day, into the following components:

- CPU idle time during screen-off;
- CPU busy time (background apps and services) during screen-off;
- CPU idle time during screen-on;
- CPU busy time (for all apps and services) during screen-on.

Note the rest of the time where the CPU is neither busy nor idle is when the CPU is in suspended state or the phone is powered off.

Figure 4(a) shows the average percentage breakdown of daily CPU time across the 2000 devices, sorted in the increasing order of the total CPU time, and Figure 4(b) shows the average of the percentage CPU time breakdowns of the 2000 devices. We make the following observations. (1) Out of the total CPU time in a day, on average, the total CPU busy time during screen-on and screen-off intervals are 9.8% and 16.7%, respectively, suggesting a significant portion of the total CPU busy time is spent during screen-off intervals running apps and services in the background. (2) On average, the fractions of the total CPU time spent in CPU idle during screen-off and screen-on intervals are 50.8% and 22.7%, respectively. The large fraction of CPU idle time compared to CPU busy time during screen-on is intuitive because the OS is often waiting for screen input but the user is idle most of the time when interacting with many apps, *e.g.,* reading web pages or emails. But the large fraction of CPU idle time during screen-off intervals is surprising, as the phone is expected to go back to sleep (*i.e.,* CPU suspended) right after waking up to perform any background activities. (3) Across the devices, while the CPU busy and idle time during screen-on and CPU busy time during screen-off stay more or less similar, the average absolute CPU idle time during screen-off increases significantly,

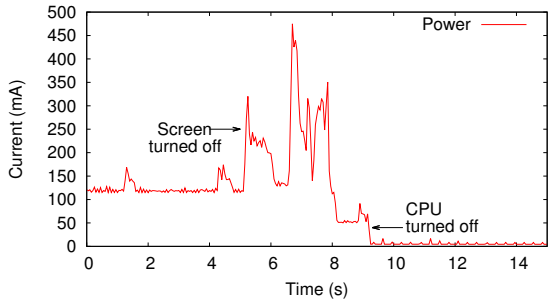


Figure 5: CPU lingers for 2-6 seconds after screen is turned off.

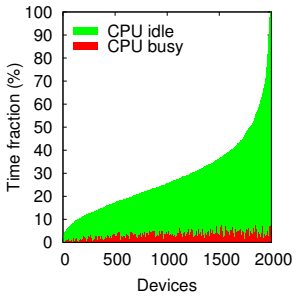


Figure 6: Total CPU busy and idle time as fractions of total interval duration of all screen-off intervals per device, after removing USB charging intervals and CPU lingering time.

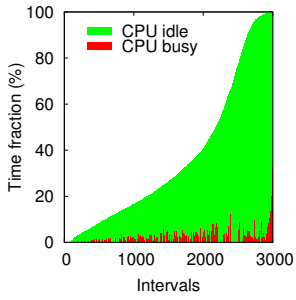


Figure 7: CPU busy and idle time as fractions of each interval duration, for 3000 sampled intervals, after removing CPU lingering time and excluding USB charging intervals.

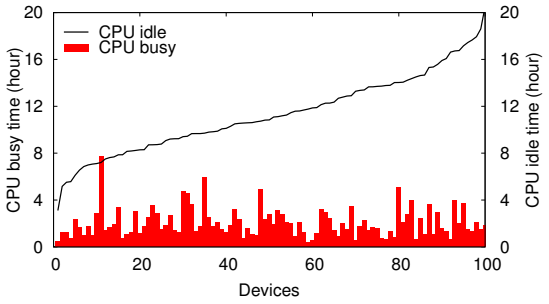


Figure 8: Cumulative daily app busy time in screen-off intervals across 100 heavily used devices, sorted by daily CPU idle time in screen-off.

from 1.4 hours on average among the least active 10% devices (on the left of Figure 4(a)) all the way up to 9.7 hours among the most active 10% devices (on the right)!

USB charging. Our quick examination of the trace revealed that the CPU was always fully awake in screen-off intervals when the phones were being charged via USB. We then verified using our own S3 and S4 phones running Android versions 4.3 and 4.3.1 that indeed when during USB charging, the CPU/SOC was never suspended.

Lingering CPU time after screen-on interval. In processing the trace, we discovered a persistent lingering CPU-on period upon entering each screen-off interval (*i.e.,* right after a screen-on interval). In our lab experiment using a powermeter as the energy source for a Galaxy S3 phone, we confirmed that the CPU stays on right after entering a screen-off interval for 2-6 seconds, as indicated by the powermeter reading shown in Figure 5. The Linux /proc shows the second lingering duration is broken down into 940 ms, 740 ms, and 4180 ms of user time, system time, and idle time, respectively. Most of the user time, 750 ms out of 940 ms, is spent in the Android's framework services, while the rest is waiting for the foreground app to pause its activities and for the other apps listening on *Intent.ACTION_SCREEN_OFF* broadcast messages. Similarly, most of the system time is also spent in various shutdown activities.

In the rest of the section, we removed all the USB charging intervals and the 6-second CPU linger time (2 seconds of busy time and 4 seconds of idle time) from all the rest screen-off intervals. Figure 4(c) re-plots the CPU time breakdown. We see that the average fraction of daily CPU time spent in CPU idle during screen-off intervals remains high at 48.7%.

3.4 Background App Activities

To understand the high CPU idle time during screen-off intervals, we plot the total CPU busy and idle time of all screen-off intervals as fractions of the total screen-off interval durations for each device in Figure 6. We see that the fractions are high – out of the 2000 devices, over 200 devices have the CPU staying up (busy or idle) for over 50% of the screen-off interval durations.

Since ultimately it is the apps and system services they depend on that wake up the device during screen-off intervals, we next zoom into the apps running on the 100 devices with the highest total fractions of CPU idle and busy time in Figure 6, *i.e.,* the rightmost 100 devices in the figure, and analyze the app activities during screen-off intervals.

(1) App busy/idle time per device. We first try to understand if there is any correlation between app busy time and app idle time in screen-off intervals. We plot in Figure 8 both quantities for the selected 100 devices. We see that there is no significant correlation between the total app busy time and the total CPU idle time across the 100 devices – on average, the CPU busy time per device in a day due to the apps in screen-off intervals is only 2.0 hours, while the corresponding CPU idle time varies from 3.1 hours all the way to 20.5 hours.

(2) App busy/idle time per interval. Next, we analyze the CPU busy and idle time in individual screen-off intervals. We uniformly sampled 3000 screen-off intervals from all 2000 devices and plotted the distribution of CPU busy and idle time in each screen-off interval as fractions of the interval duration in Figure 7. We see that the CPU busy+idle time is over 50% of the interval duration for over 27.1% of the intervals, and over 95% for 9.4% of the intervals. Further, the CPU idle time alone is over 50% of the interval duration for over 18.0% of the intervals (not directly shown).

(3) App holding wakelocks for long durations. Our investigation suggests the primary suspect for causing the large fractions of CPU idle time during screen-off intervals is inefficient or incorrect use of wakelocks in apps. Specifically, we suspect there are apps that when running in the background hold wakelocks for unnecessarily long durations causing the CPU to remain in idle for extended periods of time. To verify our hypothesis, for each device, we flag the apps that appear in more than half of the screen-off intervals when CPU hardly suspends, *i.e.,* the intervals with over 95% CPU busy+idle time, which include 9.4% all screen-off intervals as shown in Figure 7, as *suspicious apps* for that device. If one app is installed on at least 2 devices and there are more devices that have this app as suspicious than the devices that have this app but do not mark it as suspicious, then the app is marked as a "no-sleep" app [21]. Among the apps on all devices, we found 76 no-sleep apps, and 56 of them contain wakelock permissions in their manifest file. We installed 22 of them on an S4 phone running

Table 2: Summary of the hybrid power model.

Hardware component	Model type	Logged data
CPU	utilization	CPU time + CPU frequency
GPU	utilization	GPU time + GPU frequency
Screen	utilization	screen on time + brightness
WiFi	non-utilization	bytes + signal strength
3G/LTE	non-utilization	bytes + signal strength
WiFi beacon	constant	WiFi status
WiFi scan	constant	WiFi status
Cellular Paging	constant	cellular status
SOC Suspension	constant	phone status

Samsung's Android version 4.3 (Jelly Bean) in the lab for 9 hours and found that the CPU was never suspended. On further investigation, we found that one of the suspicious apps installed, Hi [5], held a wakelock, **sync*_com.android.contacts_Account*, for the entire time; indeed for the entire 9-hour duration of the screen-off interval, the CPU was busy for 1.5 hours and in the rest of the time it was in the CPU idle state. In summary, while we cannot directly confirm the reason for the large fractions of CPU idle time during screen-off intervals on the phones in the wild, we believe the likely cause is that certain apps hold wakelocks for unnecessarily long periods of time.

4. ENERGY DRAIN

In this section, we characterize the energy drain of the background activities on the 2000 devices.

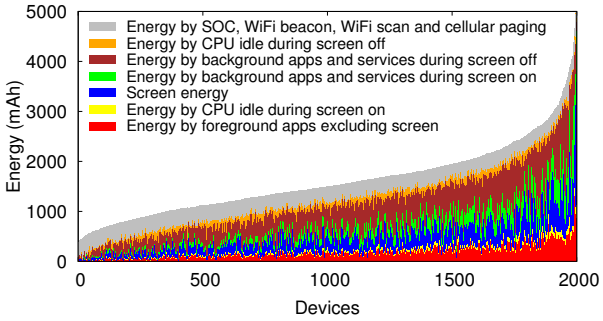
4.1 Methodology

We adopt the methodology used in [11] for measuring and analyzing energy drain on unmodified user phones. In a nut shell, the measurement is performed in three steps: (1) developing a power model that uses triggers that can be collected without any modifications to the Android framework or the kernel or rooting the phones; (2) developing and deploying an app that collects all the triggers for the power model; (3) post-processing the trace to estimate the energy drain on the phones.

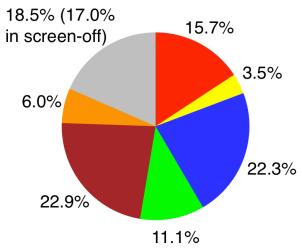
A hybrid power model. To accurately capture the power behavior of all the identified components, a hybrid utilization-based and FSM-based power model was developed in [11] which does not require modifying the Android framework or the kernel yet achieves good modeling accuracy. The model uses utilization-based modeling (*e.g.,* [24, 27]) to capture the power behavior of CPU, GPU, and screen whose power draw has linear correlation with utilization (it estimates the average screen power draw as a function of the brightness level), and FSM-based modeling (*e.g.,* [10, 17, 22, 18, 19, 12]) for wireless interfaces such as WiFi/3G/LTE. The model collects network usage in each logging interval and then estimates the send and receive system calls as triggers to drive the FSM models [20]. It further models cellular paging, WiFi beacon, and WiFi scanning as constant power draws by averaging their power spikes over time, and SOC suspension power (base power) as a constant power draw. Table 2 summarizes the power model type and triggers collected for the set of components modeled, which are selected as those that show significant power draw. As in [11], we further confirmed the component power draw are largely independent, and hence the total power draw can be approximated by summing the power draw of individual components.

We focus on Galaxy S3 and S4 phones in our measurement study, which are the most popular Android phones [3]. The details of the power models for Galaxy S3 and S4 can be found in [11].

Model validation. The hybrid model was validated in [11] using both micro-benchmarks and real apps. Using micro-benchmarks, the measured error in estimating the energy drain by individual components is within 11% of the powermeter output. On a phone with 25 popular apps installed, the measured error in estimating the energy drain of the whole phone during both screen-on and screen-off intervals is within 10% of the powermeter output.



(a) Average daily energy breakdown



(b) Daily energy percentage breakdown, average over all devices.

Figure 9: Daily energy breakdown.

4.2 Background Energy Analysis

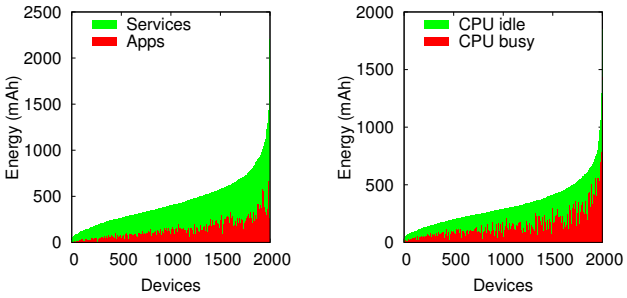
We break down the total energy per day per device among different activities as follows:

- Energy by WiFi beacon, WiFi scanning, cellular paging, and SOC base power during screen-off;
- Energy by CPU idle during screen-off;
- Energy by background services and apps during screen-off;
- Energy by background services and apps during screen-on;
- Screen energy during screen-on.
- Energy by CPU idle during screen-on;
- Energy by foreground apps excluding screen.

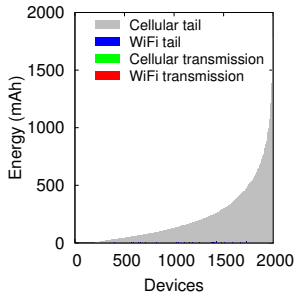
We note each app and service energy component includes both CPU and networking energy.

Figure 9 shows the energy breakdown across the 2000 devices. We make the following observations. (1) **Overall screen-on vs. screen-off:** Across the 2000 devices, on average 45.9% of the total energy drain in a day occurs during screen-off periods, including 17.0% due to SOC, WiFi beacon, WiFi scan and cellular paging. This is rather surprising, debunking the perception that when a phone is turned off, it should consume little energy. (2) **Background app/service energy in screen-off:** The background apps and services during screen-off including the induced CPU idle together contribute to 28.9% of the total energy drain[2], in contrast to the 11.1% by background apps and services during screen-on. The 28.9% gives an upper bound on how much daily energy can be saved from suppressing background activities of apps and services during screen-off intervals. (3) **CPU idle energy in screen-off:**

[2]This number is slightly higher than in [11], since the average logged screen-off duration per day in this trace (Figure 1), 18.3 hours, is longer than in that trace.



(a) App vs. service (b) CPU busy vs. idle



(c) WiFi vs. cellular, transmission vs. tail energy

Figure 10: Breakdown of background energy between apps and services, CPU busy and idle, and wireless radios.

Although on average the CPU spends 50.8% of the total CPU time in idle in screen-off (Figure 4(b)), it only drains on average 6.0% of the total energy, primarily due to the effectiveness of frequency scaling – in entering the idle state, the CPU quickly drops to the lowest frequency, *e.g.,* 384MHz on Galaxy S3, and then further into a power-saving C state which draws minimum power.

Next, we zoom into the background app and service activities during screen-off intervals, and break down the energy drain due to apps versus services, CPU busy versus CPU idle, and network energy among different radios and states. The results, shown in Figure 10, show that across the 2000 devices, on average the energy drain of background activities is roughly split 55.4% to 44.6% between apps and services, and 70.8% to 29.2% between CPU busy and CPU idle, and the vast majority of the network energy by background apps and services is spent in 3G/LTE tail energy.

5. KEY IDEA

Our analysis of energy drain of 2000 devices in the wild shows that on average 28.9% of the daily energy drain is due to apps and services running during screen-off intervals. The current solutions to curtailing this energy drain are limited. For example, iOS provides a feature called "Disable Background Refresh" [6] which prevents the apps from running in background altogether, while Android provides a similar feature called "Restrict Background Data" [1] which switches background data refresh to using WiFi links only.

In this paper, we propose HUSH, a simple yet effective screen-off energy optimizer that reduces smartphone battery drain due to background activities. The two premises behind HUSH are that (1) *background activities of apps are meant to improve user app experience but they are only useful if the user interacts with those apps in foreground some time during the next screen-on interval; and (2) the usefulness of background activities of an app is likely to be user-dependent and thus their occurrences should be personalized.*

The basic idea behind HUSH is to extract the correlation between an app's background activities, *i.e.,* whether it runs during a screen-off interval, and its foreground activities, *i.e.,* whether it would run in the foreground during the next screen-on interval. If there is a strong correlation, the background activities of the app during screen-off intervals are likely to be *useful* to the user experience. For example, for a user who checks her Facebook feeds frequently, the periodic synchronization of the Facebook app with the server during a screen-off interval reduces the response time when the user checks her Facebook during the next screen-on interval and thus enhances the user experience. On the other hand, if a user rarely checks her Facebook feeds, the background activities are likely *not useful* to the user, and disabling such periodic synchronization with the server in screen-off intervals is unlikely to affect the user experience.

Once HUSH learns the usefulness of apps' background activities during screen-off intervals, it instructs the Android scheduler to selectively avoid scheduling the execution of certain apps during screen-off intervals to save the screen-off energy.

6. BFC ANALYSIS

To validate the premise behind HUSH, in this section, we propose a metric to measure the correlation between background-foreground app activities called Background-Foreground Correlation (BFC). We then perform an extensive analysis of the BFC of all background and foreground activities on the 2000 devices. Our analysis shows that (1) the usefulness of the background activities of different apps for the same user can differ significantly; (2) the usefulness of the same app differs significantly for different users, and thus background activities of apps should be personalized to individual users. We then propose and evaluate an online algorithm that disables app background activities during screen-off intervals judiciously.

6.1 BFC Analysis

The motivation for the BFC metric is to capture the likelihood that a user will interact with an app during a screen-on interval after it had any background activities during the preceding screen-off interval. Formally, for a given app k, let B_k denote a given sequence of all screen-off intervals in time order during which app k was active. For each interval b in B_k, we define binary function $X_k(b) = 1$, if app k ran in foreground during the screen-on interval following b, and $X_k(b) = 0$, otherwise. The BFC metric of app k for B_k is calculated as

$$\text{BFC}_k(B_k) = \sum_{b \in B_k} X_k(b)/|B_k| \qquad (6.1.1)$$

The BFC of an app is thus a value between 0 and 1. The lower the BFC value, the weaker the correlation between the app's background activities during screen-off intervals and its foreground activities during subsequent screen-on intervals, and hence suppressing the app's background activities during screen-off intervals will most likely not affect the user experience with the app.

Limitations. The BFC metric does not capture well the usefulness of the class of apps that primarily run in the background, such as file-syncing apps or health monitoring apps. For example, Dropbox is used by users to upload files to the Dropbox site and this may happen on a scheduled frequency which may occur during screen-off intervals only, and the user may only infrequently access the Dropbox app itself during screen-on. In this case, the BFC metric will be low yet the background upload activities are considered useful to the user. For such apps, HUSH provides a feature that allows users to whitelist them to bypass the suppressing procedure.

BFC of the apps of the same user varies. We first measure the BFC of different apps on the same device. We randomly picked

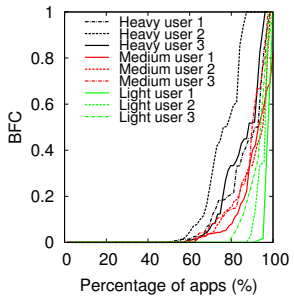


Figure 11: BFC of all apps across 3 light users, 3 median users, and 3 heavy users.

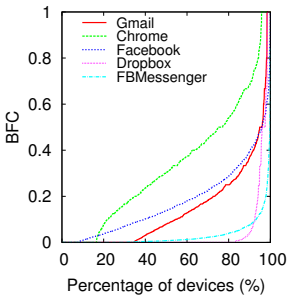


Figure 12: BFC of top 5 most popular apps across users

Table 3: Top 5 popular apps in the 2000-user trace.

App	Popularity
Gmail	1718
Chrome	1601
Facebook	1462
Dropbox	1259
FB Messenger	1257

3 light users, 3 median users and 3 heavy users according to daily screen-on time of Figure 1. Figure 11 shows the BFC for all the apps during the trace period, 27.9-day long on average, for each of the 9 users. We make several observations. First, the heavy users have the highest BFC, followed by the median users and by light users. This is intuitive as heavy users spend more time playing apps in foreground, making more background app updates useful. Second, for all 9 users, nearly 60% of the apps have a BFC value of 0, suggesting that many apps are not accessed at all by users after they performed some background activities. This is often the case where the user used an app for a short while, and then lost interest and stopped using it. In other cases, an app's background activities may not be a driving factor (*e.g.,* the app did not generate any notifications) for the user to access it, and thus the user experience is likely unaffected if its background activities are suppressed. Third, for the remaining 30%–40% of the apps on each device, their BFC values vary significantly, all the way from 0 to 1, suggesting the BFC is highly app-dependent.

BFC of an app varies with users. Next, we measure the BFC of an app across different users. We pick the top 5 most popular apps in our 2000-user trace, shown in Table 3, and calculate the BFC for each of them on the 2000 devices. The results are shown in Figure 12. We make several observations. First, the BFC of all 5 apps indeed vary significantly, ranging between 0 to 1, suggesting it is highly user-dependent. Second, even for popular apps such as Gmail, the BFC is zero for 35% of the users, who will benefit from HUSH which would suppress the app from running during screen-off intervals and save battery, without experiencing any adverse user experience. Third, interestingly, while Dropbox indeed has a BFC of zero for about 80% of the users, its BFC is large for the remaining 20% users, reaching 1 for 1.1% of the users. The detailed trace shows for those devices, the user checks Dropbox in foreground every time after it synchronizes with its server in a screen-off interval.

Stability of BFC. Next, we measure whether the BFC of a given app on a given user's phone is stable over time. For each user trace, we divide the trace duration into non-overlapping windows of 24 hours each, and calculate the BFC for each window for each app that has at least 24 daily background activities (on average one per

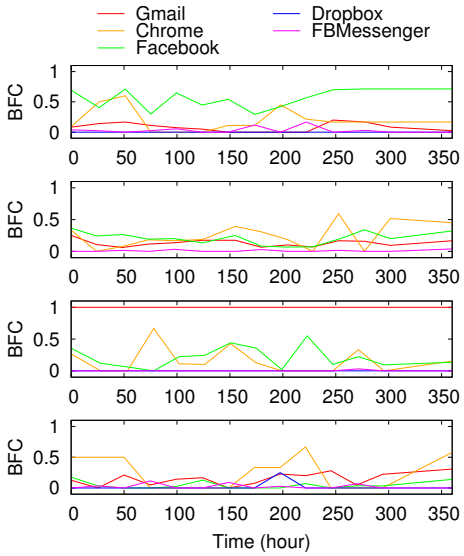


Figure 13: Stability of the BFC of 5 apps on 4 selected devices.

hour). Figure 13 plots the BFC values across all the windows for the same 5 apps studied in Figure 12 and 4 out of the 9 devices in Figure 11 (the second user has no Dropbox). Further, we pick top 25 apps across the devices, calculate the standard deviation of the BFC values for all the windows for each app on each device that has it, and plot in Figure 14 the distribution of standard deviation values per app across all the devices. We observe that (1) the BFC for the same app on the same device is fairly steady (*e.g.,* FBMessenger) or changes slowly (*e.g.,* Facebook), and (2) while the BFC for the same app may vary sharply across users, it is fairly stable for individual users.

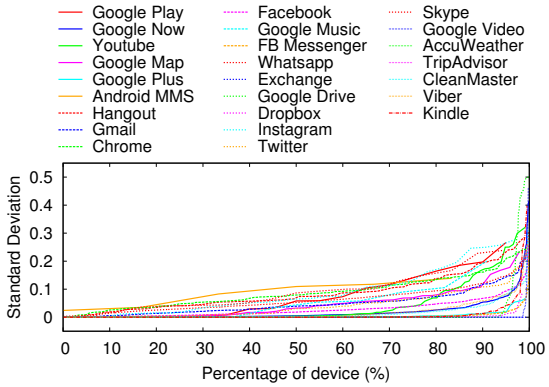


Figure 14: Standard deviation of the BFC of top 25 apps on all devices.

In summary, the results from the above three measurements suggest that the BFC value is both user-dependent and app-dependent but is fairly steady for the same app and same user, and therefore should be and can be learned on a per-app and per-user basis.

6.2 Prediction-based Online Algorithm

The BFC analysis above motivated us to develop a prediction-based online algorithm for suppressing app background activities to reduce the background energy drain. The algorithm uses an exponential moving average to continuously update the BFC for each

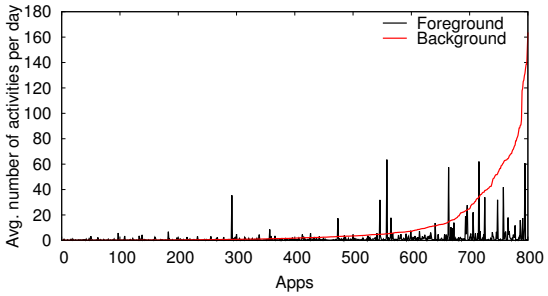


Figure 15: Average numbers of foreground and background activities per day of top 800 apps on the 2000 devices.

app on each device, as follows:

$$\text{BFC}_k(i) = \beta \cdot \text{BFC}_k(i-1) + (1-\beta) \cdot X_k(i) \qquad (6.2.1)$$

where $\text{BFC}_k(i)$ denotes the BFC updated at the end of the screen-on interval after screen-off interval i which had background activities of app k, and $X_k(i)$ records if app k ran in that screen-on interval.

In the following screen-off interval $i+1$ when app k attempts to wake up to perform background activities, the algorithm compares the app's current BFC(i) to a predetermined cutoff value α: the background activity is suppressed if $\text{BFC}_k(i) \leq \alpha$. However, $\text{BFC}_k(i+1)$ is updated as usual regardless if the background activities during interval $i+1$ were suppressed or not.

6.3 Choosing Parameters

The online algorithm above has two key parameters: coefficient β in the exponential moving average calculation and cutoff α for suppressing app background activities. Both parameters directly affect the suppression operations of the algorithm.

To gain insight into how to choose the parameters, we first show the average numbers of foreground and background activities per day of the top 800 apps on the 2000 devices (*i.e.,* with over 15 users) in Figure 15. We find 22% apps have more than 10 background activities per day, 10% apps have more than 36 background activities per day, and 77% of apps have more daily background activities than daily foreground activities. We further study the 300-hour long background and foreground activity sequences of 10 popular apps running on a dozen devices, and calculated their BFC under β values of 0.1, 0.5, and 0.9. Figure 16 shows the results for 6 representative app-device combinations. Based on these background activity patterns, we can group apps into three cases:

- Case 1: there are few foreground activities and hence many background activities per foreground activity (*e.g.,* Google Now, and Gmail);
- Case 2: there are many foreground activities and hence few background activities per foreground activity (*e.g.,* Facebook-User 2, Whatsapp-User 2);
- Case 3: the app contains alternating phases with few and many foreground activities (*e.g.,* Facebook-User 1, Whatsapp-User 1).

Notice the same app (*e.g.,* Whatsapp) can fall into different cases on different devices because of different user behavior.

The above categorization of app background activities can be intuitively explained as follows. All the background activities that we see are push-based. Pushed information are either *time-sensitive* or *time-insensitive*. If the information is time-sensitive, the app typically notifies the user to use the app and the user is likely to open the app. For example, chats and certain types of posts (*e.g.,* a friend directly interacts with a user by sending messages or writing to her timeline) are generally considered time-sensitive and produce notifications, and hence Whatsapp and Facebook (User 2) fall into Case 2. In contrast, the information fetched by Google Now in background, *e.g.,* recommended news and weather forecast, seldom are time-sensitive or generate notifications, and rarely prompt the user to use the app. Hence Google Now falls into Case 1. The information fetched by Facebook may consist of a mix of general friend's posts which are not time-sensitive, and time-sensitive posts mentioned above. Thus, Facebook can also fall into Case 3, *e.g.,* Facebook - User 1.

Picking the β value. In Case 1, since the app has few foreground activities, the app's BFC should stay low, and hence a large β value is preferred as it favors the steady-state (low) BFC; with a small β, transient foreground activities would cause the BFC to jump up significantly, as shown in Figure 16(c)(d).

In Case 2, there are many foreground activities and also background activities, and hence $X_k(i)$ oscillates between 0 and 1 frequently. Since many background activities are useful, background activities should not be suppressed and hence BFC should not go too low. In this case, a larger β is preferred as using a small β value would give too much weight to the most recent $X_k(i)$ which can cause the BFC to fluctuate between close-to-0 and close-to-1 values, as seen in Figure 16(b)(f).

Finally, in Case 3, the app transitions between strong and weak correlation phases, a small β is preferred since it allows the BFC to quickly adjust to the steady-state value in the new phase, as shown in Figure 16(a)(e).

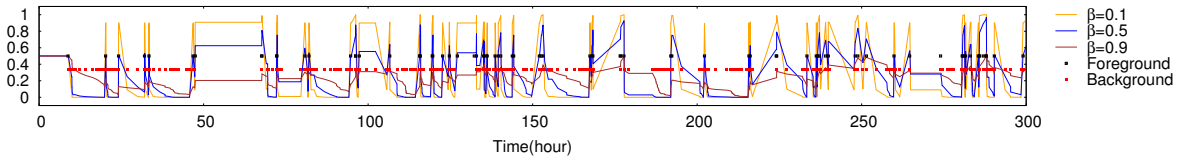
In summary, the three cases prefer different values of β. We pick β=0.5 to strike a balance between the difference preferences. As shown in Figure 16, using β=0.5 keeps the BFC sufficient low for Case 1 and sufficiently high for Case 2, and adapts the BFC quickly for Case 3. The BFC curves using β=0.5 in the figure also suggest that using a low value of α in the range of 0.1-0.2 will suppress most background activities appropriately.

6.4 Trace-driven Evaluation

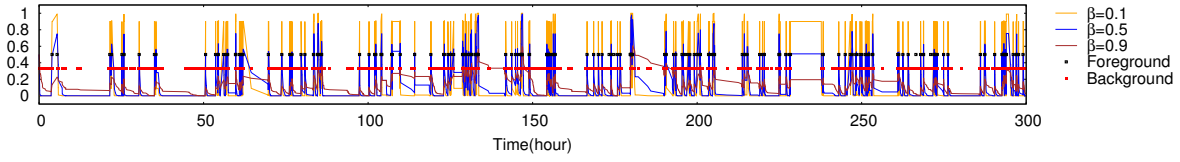
We now evaluate the effectiveness of our online BFC-based algorithm for suppressing background activities to save screen-off energy using trace-driven simulation, using our 2000-user trace. The *energy saving* is defined as $\frac{E_b - E_a}{E_b}$, where E_b and E_a are the total energy on each device before and after our suppression algorithm is turned on.

Methodology. When we suppress background activities in screen-off, we assume their networking is completely avoided, *i.e.,* not delayed to be performed during the next screen-on activity of the app, which potentially can increase the screen-on part of the app energy. Since we do not know the causality between app and system service activities, we break down the total energy drain due to system services in each logging interval and attribute them to the apps that ran in that interval proportionally to their CPU busy time. Similarly, we break down the CPU idle time and hence energy in each interval and attribute them to the apps that ran in that interval proportionally to their CPU busy time. When an app background activity is suppressed, the associated service and CPU idle energy drain are also avoided.

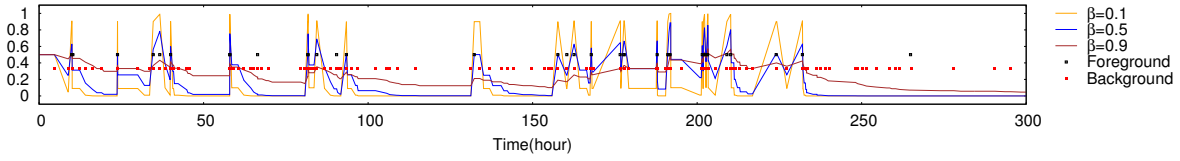
Since we have networking statistics per 5-second logging interval, the total network energy calculation in Section 4 already took into account app background networking activities in the same 5-second interval would share network tail energy. Similarly, when an app background activity in a screen-off interval is suppressed, its share of network energy is deducted appropriately, *i.e.,* if other apps and services had networking activities in the same 5-second interval, the tail energy is not deducted.



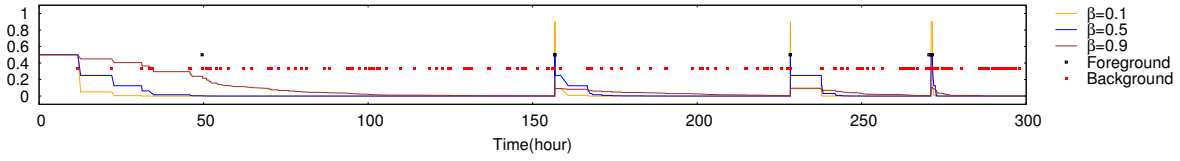
(a) Facebook - User 1 (Case 3)



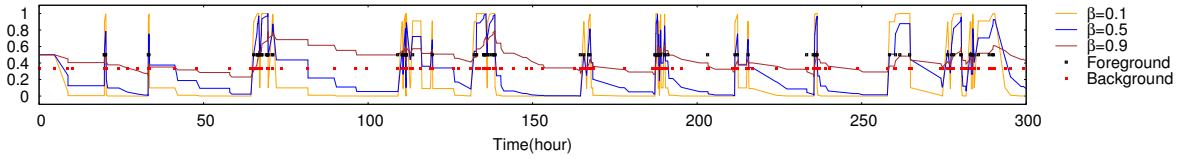
(b) Facebook - User 2 (Case 2)



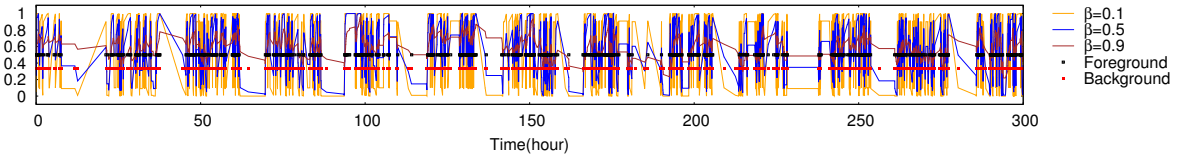
(c) Gmail (Case 1)



(d) Google Now (Case 1)



(e) Whatsapp - User 1 (Case 3)



(f) Whatsapp - User 2 (Case 2)

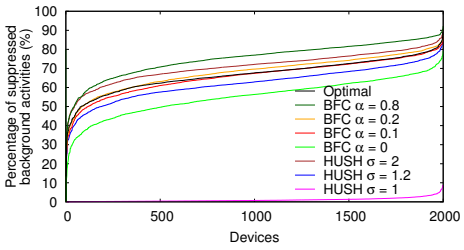
Figure 16: Background/foreground activities and online BFC calculation under three β values. BFC is initialized to 0.5.

Staleness. Since our practical algorithm uses an app's past BFC to predict its future BFC, it can mispredict and suppress background activities that turned out to be useful, *i.e.,* the next screen-on interval had foreground activities of the app. To quantify such mispredictions, we define the *staleness* for each app foreground activity as the elapsed time since the last background activity, or the last foreground activity of the app, whichever is closer in time. The *staleness of an app* on a device is then defined as the average staleness of all its foreground activities on that device. We note under this definition, even without any suppression, the staleness of an app is not zero, but an incorrect suppression increases the staleness.

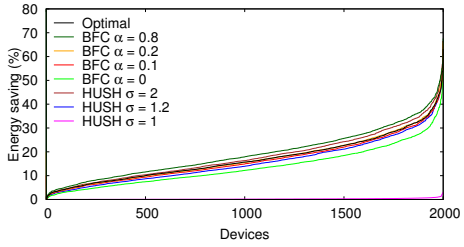
Optimal energy saving. Before presenting the results for our online prediction-based algorithm, we calculate the *optimal energy saving*, the energy saving of an optimal suppression algorithm that knows the future and disables all background activities of any app that ran in screen-off interval b but did not run in the screen-on interval immediately following screen-off interval b. In doing so, the screen-off energy would be reduced by 14.1%–90.0% across the 2000 devices, with an average of 55.6%, which translates into 0.2%–63.1% total daily energy saving, with an average of 17.1% and a median of 15.7%, as shown in Figure 17(b).

Energy saving. We evaluate the algorithm with different threshold α values, which control the trade-off between screen-off energy reduction with app staleness. For example, an α value of 0 is the most conservative – an app's background activities are suppressed only if its current BFC metric is 0, which also leads to the least amount of energy saving.

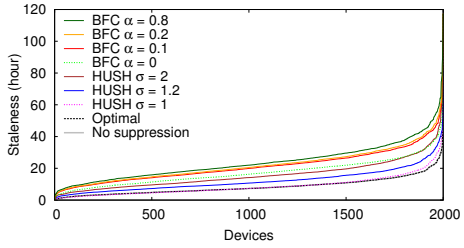
Figures 17(a)(b) show the percentage of background activities suppressed and the total energy saving across the 2000 devices as α varies between 0 and 0.8 and $\beta = 0.5$. We observe that using a small α value of 0.1, significant amount of background energy can already be reduced across the 2000 devices, with the average, the 10th, 50th, and 90th percentile total energy saving being 16.4%, 6.0%, 14.9% and 29.4%, respectively. Additional energy saving with more aggressive suppression, *i.e.,* by relaxing α all the way to 0.8, is small, with the average, the 10th, 50th, and 90th percentile



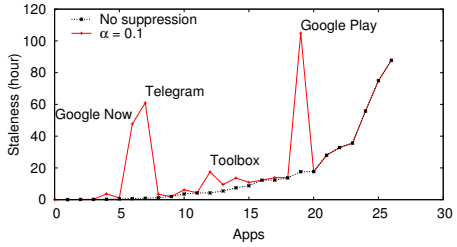
(a) Percentage of suppressed background activities



(b) Energy saving



(c) Average staleness of apps



(d) Staleness per app on one device from BFC-based algorithm (with 2.5x staleness for α=0.1)

Figure 17: Results from running the BFC-based algorithm and the HUSH algorithm across the devices, β=0.5.

total energy saving being 19.3%, 7.3%, 17.9%, and 33.2%, respectively. This is because most of the energy saving comes from Case 1 apps, whose BFC will almost always stay below 0.1.

Figure 17(c) shows the corresponding app staleness under the different α values. We see that suppressing background activities increases the app staleness from on average 2.5x with $\alpha = 0.1$ to 3.1x with $\alpha = 0.8$. The above trade-off between energy saving and app staleness suggests choosing an α value of 0.1 strikes a good balance between the two metrics. To understand the seemingly large staleness increase, we picked the device that corresponds to 2.5x increase under α=0.1 in Figure 17(c), device 1400, and plot the staleness of the individual apps on it before and after running the suppression algorithm, in Figure 17(d), sorted by staleness without suppression. We see that for most apps, the staleness stays the same, suggesting the suppression of their background activities was accurate; only 4 out of 27 apps have significantly increased staleness, which contributes to the 2.5x average staleness of the device.

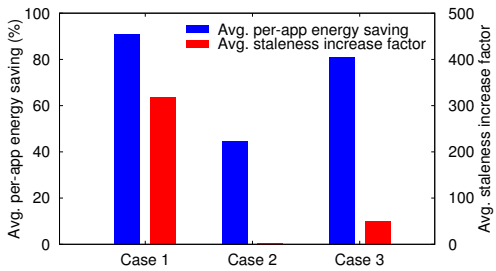


Figure 19: Average energy saving and staleness increase factors of apps in the 3 cases on the 2000 devices.

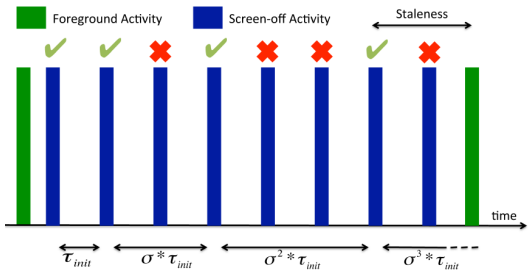


Figure 20: The HUSH suppression algorithm.

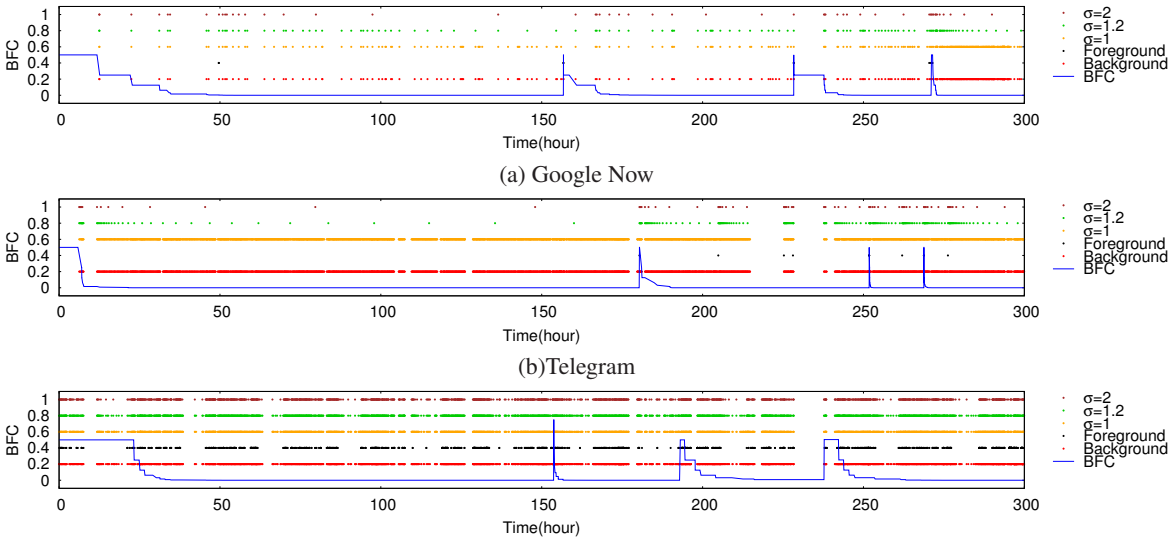
Algorithm refinement. We dig deeper into the 4 apps (Google Now, Telegram, All-In-One Toolbox, Google Play) with large staleness and plot their 300-hour long background and foreground activity sequences in Figure 18 (with Google Play omitted due to page limit). We observe these apps are all Case 1 apps; they rarely run in foreground, and hence when background activities in between two adjacent foreground activities are suppressed, the screen-off energy is reduced significantly but the staleness also skyrockets.

To generalize this finding, we classify the user-app combinations into each case using a simple algorithm as follows. We assign a user-app combination as Case 1 (Case 2) if for 80% of the foreground activities there are more (less) than 5 background activities until the next foreground activity, and Case 3 otherwise. We draw the average per-app energy saving and the ratio of staleness (after and before suppression) in Figure 19 under α=0.1 for the three Cases of apps and observe that Case 1 apps indeed have the most energy savings as well as the highest staleness increase.

The BFC-based suppression algorithm strictly follows an all-or-none policy; once the BFC decays below α, it suppresses *all* background activities in the subsequent screen-off intervals leaving the staleness grow boundlessly until the next foreground activity. We observe that the staleness of Case 1 apps can be significantly reduced while retaining their energy saving, by relaxing the strictness of the suppressing policy to include an exponential backoff and omitting the need to maintain BFC calculation, as shown in Figure 20. We denote this new algorithm as HUSH.

HUSH does not maintain the BFC calculation since for Case 1 apps the BFC between two foreground activities will be very low that it cannot predict the next foreground activity anyway. Instead, it simply maintains and tunes a single threshold time parameter (τ) for each app; an app's next screen-off activity is allowed only if its previous background activity happened τ earlier, and is otherwise suppressed. Whenever HUSH allows a screen-off activity, τ is multiplied by a scaling factor σ (>1), making screen-off activities less and less frequent. However, when an app comes to foreground, its τ is reset back to τ_{init}. Figure 18 shows how HUSH suppresses the background activities for the 3 apps for different values of σ.

Compared to the BFC algorithm, HUSH just allows $O(log_\sigma(K))$ screen-off activities where K is the total screen-off activities be-



(a) Google Now

(b)Telegram

(c) All-in-one Toolbox

Figure 18: Original background/foreground activities, and background activities after suppression by the BFC algorithm and by the HUSH algorithm, for 3 apps with large staleness.

tween two foreground activities with interval D, while bounding app staleness to be about $\frac{\sigma-1}{\sigma} \cdot D$, assuming the background activities are equally spaced. Thus, it turns out HUSH works well not only for Case 1 apps, but also for Case 2 (and hence Case 3 apps), for which the background activities will be barely suppressed as desired, if τ_{init} is set to an appropriate value – we set it to 1 minute.

We also plot the average energy savings and staleness of apps across 2000 devices in Figure 17 for HUSH with different values of σ. We see the HUSH algorithm does not incur large staleness and yet is able to achieve similar energy savings as the BFC algorithm. With $\sigma = 1.2$, the average energy saving across the 2000 devices is 15.7% compared to 16.4% for BFC with $\alpha = 0.1$, and the staleness increase is 1.3x on average compared to 2.5x in BFC with $\alpha = 0.1$.

7. HUSH SCREEN-OFF ENERGY OPTIMIZER

The Android framework exposes to apps several services such as Location Manager, Alarm Manager, and Power Manager, which run with higher privilege than apps in a separate process, *system_server*. Android services typically use the publish-subscribe pattern where apps subscribe to updates (*e.g.,* location updates) from the services (*e.g.,* Location Manager), and services then *invoke* app callbacks upon the subscribed event (*e.g.,* user location changed), via either the synchronous RPC mechanism *Binder*, or the asynchronous IPC message passing mechanism *Intent*.

Thus, to suppress the default app background activities, HUSH tracks app foreground and background activities and intercepts app invocations by framework services during screen-off intervals by either allowing or rejecting them.

7.1 HUSH Design

Figure 21 shows the architecture of HUSH design. The main algorithm of HUSH is implemented inside *BatteryStatsImpl*, where Android maintains runtime statistics of the whole system and for each individual app such as the total screen-on time, total number of wakeups caused by each app, and total time a wakelock was held by each app. Other HUSH components directly interact with the *BatteryStatsImpl* module. *ActivityManagerService* and *BatteryStatsImpl.Uid.Pkg.Serv* notify the central components of any foreground and background app activities, respectively.

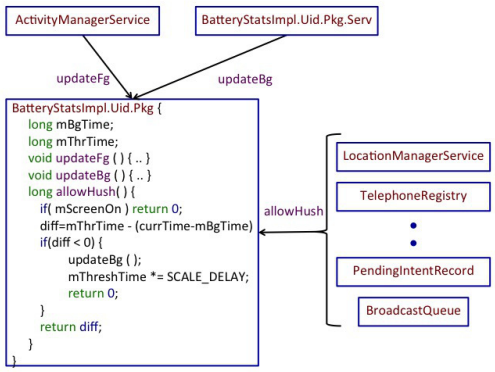


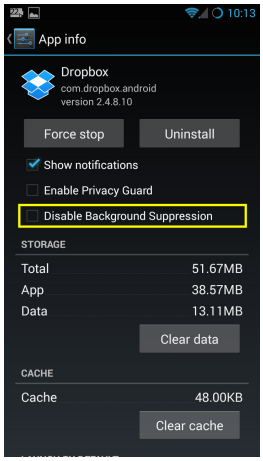
Figure 21: Architecture of HUSH.

Most framework services including *AlarmManagerService* [9] and *GoogleCloudMessaging* [8] send data to apps or simply wake up apps' *BroadcastReceiver* via lightweight asynchronous IPC mechanism – *Intent*. *BroadcastQueue* now only sends broadcasts when *allowHush* returns true.

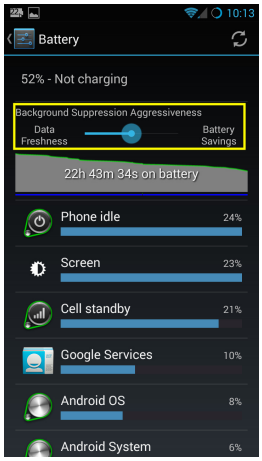
Other framework services such as *LocationManagerService* use Android's synchronous RPC mechanism– *Binder*. We directly edit such framework services to only make the remote call if *allowHush* returns true. HUSH does not intercept *Binder* remote calls since upon *allowHush* failure, HUSH cannot (1) throw an exception, since any remote exceptions are not sent back to the caller and hence a caller generally does not expect an exception, or (2) return a garbage value as it can break the caller's code logic.

We note that none of these changes harm user-facing app components. *allowHush* allows all requests when the screen is on and when an app is perceptible to the user in screen-off, such as music streaming apps and navigation apps. Users can also whitelist apps and vary the aggressiveness of HUSH by changing the decay parameter σ in Android settings, as shown in Figure 22.

In total, HUSH adds and modifies 450 and 272 lines of code, respectively. The HUSH source code was released in March 2015 and can be downloaded at *github.com/hushnymous*.



(a) Disable background suppression for selected apps.



(b) Control background suppression aggressiveness.

Figure 22: Hush allows users to control its behavior.

Table 4: Statistics of HUSH running on two Samsung Galaxy S3 phones with Android Jelly Bean for 6 days.

	User-1		User-2	
Number of installed apps	73		52	
Daily screen-on intervals	85		29	
Daily screen-on time (min)	82.35		49.95	
Daily suppressions by HUSH	4400		5543	
	Android	HUSH	Android	HUSH
Daily CPU busy time (min)	164.2	97.04	60.81	27.24
Maintenance power (mA)	12.76	12.76	12.12	12.12
Avg. screen-off power (mA)	15.57	5.27	3.19	2.18
Avg. screen-on power (mA)	316.8	323.5	271.4	273.0
Overall avg. power (mA)	45.50	36.34	27.32	18.99

7.2 Evaluation

We evaluate HUSH on two Samsung Galaxy S3 phones. Both phones ran Android with HUSH for 3 days and unmodified Android for the other 3 days. We installed and whitelisted eStar in HUSH so we can use it to monitor the usage pattern and energy drain rate of the phones.

Table 4 shows the usage pattern for the two users. User-1 uses his phone heavily having 73 apps installed on the phone and keeps the screen on for an average of 82 minutes daily. While User-2 has 52 apps and turns on the screen for 50 minutes daily. On average, HUSH suppressed 4400 and 5543 app background activities on the two phones daily, which reduced the daily CPU busy time by 1.69x and 2.23x. The maintenance power, which includes the normalized power draw by cellular paging, WiFi beacon, WiFi scanning, and SOC in suspension, differ slightly between the two phones which were connected to WiFi for different durations. HUSH significantly reduced the average screen-off power, *i.e.,* screen-off energy normalized by screen-off duration, by 2.95x and 1.46x, while the average screen-on power is slightly increased, leading to an overall average power reduction of 1.25x and 1.44x for the two users, respectively.

Next, we sort all the apps by the ratio of the number of times they were allowed to perform screen-off activity to the number of times they were suppressed by HUSH. We only consider apps that tried to perform screen-off activities at least 5 times.

Table 5 shows the top 5 most allowed and most suppressed apps by HUSH on the two phones. The majority of apps in both allowed and suppressed tables are from the Communication Google Play category which comprises of IM clients, email clients and browsers. The top most allowed app for User-2 is Sogou input which is a keyboard app for typing Chinese characters and hence comes to foreground often. Sogou input syncs with its server during screen-off to get the current popular words and phrases.

Table 5: Most allowed/suppressed apps by HUSH.

User-1		User-2	
App	Allow-deny ratio	App	Allow-deny ratio
Most allowed			
Whatsapp	0.338	Sogou input	2.000
Google Hangout	0.333	iReader	0.473
Google Plus	0.182	Chrome	0.403
Meditation Helper	0.182	Skype	0.354
Gmail	0.142	Gmail	0.235
Most suppressed			
Hydro Coach	0.041	Evernote	0.000
Cover Screen	0.042	Zaker	0.020
Email	0.048	QQ	0.035
Chrome	0.059	Wechat	0.091
Aviate	0.065	Weibo	0.097

We found that both users keep more than one app that perform the same function, *e.g.,* messaging. However, out of these, the users typically use one app to perform one function. Hence exactly one app performing similar functionalities appears in the most allowed app list; the alternates are in the most suppressed list, because they hardly ran in foreground, with the sole exception of Whatsapp and Google Hangout. For User-1, Gmail is in the most allowed app list, but Email is the most suppressed list. In IM apps for User-2, Skype is in the most allowed list but Wechat and QQ are in the most suppressed list. Similarly, in news & magazines category, iReader is among the most allowed but Zaker is among the most suppressed.

Limitations. Although we released the source code of HUSH in March 2015, we have had limited experience with evaluating HUSH on real users' phones so far. The challenge stems from the difficulty in performing controlled experiments, *i.e.,* to subject a user's phone with and without running HUSH to the same user app usage and external conditions. A related limitation is that we have not performed any human subject study, *i.e.,* to measure whether running HUSH on a user's phone affects the user experience. We plan to study ways of overcoming these challenges in our future work.

8. CONCLUSIONS

In this paper, we performed to our knowledge the first in-depth large-scale measurement study of background activities on smartphones in the wild and their impact on battery drain. Our analysis shows that across the 2000 Galaxy S3 and S4 devices, on average 45.9% of the total energy drain in a day occurs during screen-off intervals, and the background apps and services and induced CPU idle time during screen-off together contribute to 28.9% of the total energy drain. We then proposed the BFC metric to measure the usefulness of background activities and showed the usefulness of background activities is highly app-dependent and user-dependent. Finally, we presented a screen-off energy optimizer called HUSH that automatically identifies and suppresses background activities of individual apps during screen-off periods that are not useful to the experience of the user. We show HUSH can save the screen-off energy across the 2000 smartphones by 15.7% on average.

Acknowledgment. We thank the anonymous reviewers and our shepherd for their constructive comments which helped to improve this paper. This work was supported in part by NSF grant CCF-1320764 and by Intel.

9. REFERENCES

[1] Android background app usage restriction.

[2] Android service. http://developer.android.com/reference/android/app/Service.html.

[3] AppBrain, top android phones. www.appbrain.com/stats/top-android-phones.

[4] estar energy saver. https://play.google.com/store/apps/details?id=com.mobileenerlytics.estar.

[5] Hi. https://play.google.com/store/apps/details?id=com.didirelease.view.

[6] ios background app refresh.

[7] ios background execution.

[8] Receive a message. developer.android.com/google/gcm/client.html#sample-receive.

[9] Using pending intent. developer.android.com/reference/android/app/AlarmManager.html#set(int, long, android.app.PendingIntent).

[10] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proc of IMC*, 2009.

[11] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. In *ACM SIGMETRICS*, 2015.

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

[13] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proc. of IMC*, 2010.

[14] 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 Mobisys*, 2012.

[15] J. Huang, F. Qian, Z. M. Mao, S. Sen, and O. Spatscheck. Screen-off traffic characterization and optimization in 3g/4g networks. In *IMC*, 2012.

[16] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance difference on smartphones. In *Proc. of Mobisys*, 2010.

[17] C.-Y. Li, C. Peng, S. Lu, and X. Wang. Energy-based rate adaptation for 802.11n. In *Proc. of ACM MobiCom*, 2012.

[18] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Proc. of ACM MobiCom*, 2012.

[19] 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 EuroSys*, 2012.

[20] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system-call tracing. In *Proc. of EuroSys*, 2011.

[21] A. Pathak, A. Jindal, Y. C. Hu, and S. Midkiff. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of Mobisys*, 2012.

[22] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proc. of Mobisys*, 2011.

[23] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *OSDI*, 2012.

[24] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *MICRO*, 2009.

[25] J. Sommers and P. Barford. Cell vs. wifi: On the performance of metro area mobile connections. In *IMC*, 2012.

[26] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *Proc. of IMC*, 2011.

[27] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc. of CODES+ISSS*, 2010.