PowerForecaster: Predicting Smartphone Power Impact of Continuous Sensing Applications at Pre-installation Time

Chulhong Min[1], Youngki Lee[2], Chungkuk Yoo[1], Seungwoo Kang[3], Sangwon Choi[4], Pillsoon Park[5], Inseok Hwang[6], Younghyun Ju[7], Seungpyo Choi[1], Junehwa Song[1]

[1]School of Computing, KAIST, [2]School of Information Systems, Singapore Management University, [3]Computer Science and Engineering, KOREATECH, [4]Information and Electronics Research Institute, KAIST, [5]Division of Web Science Technology, KAIST, [6]IBM Research – Austin, [7]Naver Labs

[1,5]{chulhong, ckyoo, spchoi, pillsoon.park, junesong}@nclab.kaist.ac.kr, [2]youngkilee@smu.edu.sg, [3]swkang@koreatech.ac.kr, [4]sangwonc@kaist.ac.kr, [6]ihwang@us.ibm.com, [7]younghyun.ju@navercorp.com

ABSTRACT

Today's smartphone application (hereinafter 'app') markets miss a key piece of information, power consumption of apps. This causes a severe problem for continuous sensing apps as they consume significant power without users' awareness. Users have no choice but to repeatedly install one app after another and experience their power use. To break such an exhaustive cycle, we propose PowerForecaster, a system that provides users with power use of sensing apps at pre-installation time. Such advanced power estimation is extremely challenging since the power cost of a sensing app largely varies with users' physical activities and phone use patterns. We observe that the time for active sensing and processing of an app can vary up to three times with 27 people's sensor traces collected over three weeks. PowerForecaster adopts a novel power emulator that emulates the power use of a sensing app while reproducing users' physical activities and phone use patterns, achieving accurate, personalized power estimation. Our experiments with three commercial apps and two research prototypes show that PowerForecaster achieves 93.4% accuracy under 20 use cases. Also, we optimize the system to accelerate emulation speed and reduce overheads, and show the effectiveness of such optimization techniques.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-based Systems**]: Real-time and embedded systems

Keywords

Power impact; Sensing applications; Pre-installation; Smartphone

1. INTRODUCTION

Today's smartphone app markets help users select desirable apps, providing diverse information such as features, screenshots, and user comments; however, they are missing a key piece of information: *power consumption* by an app. Users can only find out whether an app is power hungry after they install and use it. They count on such experiential perception to decide whether they keep the app or how long they use it. However, continuous sensing apps [25][28] make such experiential power control no longer effective. They continuously drain battery in the background without a user's explicit awareness. For example, a pedometer app, Accupedo [1], consumes up to 200 mW of additional power, causing an early shutdown of a user's phone.

SenSys '15, November 1-4, 2015, Seoul, South Korea
© 2015 ACM. ISBN 978-1-4503-3631-4/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2809695.2809728

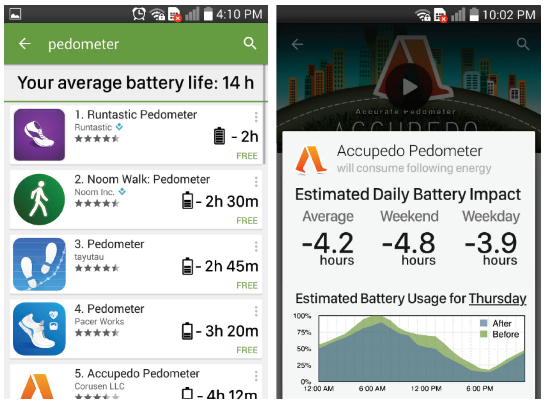


Figure 1. Power impact at pre-installation time

What if an app market provided the estimated power use of a sensing app? Given such information, users could make better informed decisions, even prior to installing the apps. Users are relieved of exhaustive trial and error, can decide judiciously to install a certain app, and may be less embarrassed with rapid battery depletion if they decide to accept the expected battery cost.

It is not straightforward to realize such a function in reality. A trivial way is for an app developer to post the average power of an app for common use cases. However, we noticed that such a reported power number would not be accurate for many individual users. The deviation mainly results from that the power use of a sensing app varies noticeably depending on users' physical activities, phone use, and other environmental factors. The error becomes even larger when the app applies various optimization techniques such as triggering power-hungry GPS with low-power accelerometers (See Section 2.2).

In this paper, we propose *PowerForecaster*, a system to provide an *instant*, *personalized* power estimation of a sensing app *at pre-installation time.* Figure 1 shows the two mockup screenshots that users would see in an app market when the system is integrated. PowerForecaster aims at enabling a number of unique user experiences. First, it provides power estimation at *pre-install time*, removing the hassle of installing and using apps one after another until users find power-efficient apps. Second, estimation is highly *personalized* to reflect an individual user's activity and phone use patterns, achieving higher estimation accuracies. Third, the system supports a wide variety of sensing apps without requiring any changes in the apps or additional information from developers.

To realize the unique features above, we take a *trace-driven emulation* approach. The key idea is (1) to pre-collect sensor and device usage traces on a user's real phone, which represent a user's physical activities and phone use patterns, respectively, and (2) to build a *power emulator* that assesses expected power use of a sensing app based on the pre-collected traces. The emulator executes the target sensing app over the sensor traces and tracks changes in the power states of key hardware components. At the end, it maps each power state into a power number and calculates total power consumption by aggregating all the power numbers caused by hardware use. In addition, it tracks the hardware use of other apps by replaying the device usage traces for accurate net power estimation. Note that a sensing app runs continuously and shares hardware with other apps. In this way, the system estimates power impact of the target app, accurately reflecting user behavior.

We address a number of technical challenges to make our approach feasible. Most importantly, we designed our own Android power emulator with significant extension to the original Android emulator to support various sensors, trace replay, and power tracking. The Android emulator [2] does not track power states of hardware components or emulate common sensor devices such as accelerometers. Moreover, it does not feed pre-collected sensor traces to the emulator, thereby precluding the possibility of emulation over users' real traces. Other existing emulators also [17][54] do not support power emulation of sensing apps.

We further optimize PowerForecaster to make its emulation fast and its collection of traces power efficient. First, it is needed to use longer user behavior traces for reliable power estimation, but this makes emulation very time consuming. The emulation by default takes as much time as the length of original traces. Advance emulation is challenging due to enormous combinations of users, sensing apps, and their version updates. PowerForecaster achieves fast emulation by (a) fast-forwarding replays in a way to capture changes in power states only and (b) parallelizing replays on multiple emulator instances. Our evaluation shows that it emulates 18-hour long traces within half of a minute, with a small error (6-7%). Second, it incurs nontrivial power overhead to collect sensor traces on a user's phone. From our empirical experiences, we developed a balanced duty-cycling policy to minimize necessary data collection while keeping power estimation accuracy high. Users do not need to repetitively collect sensor traces for various sensing apps. Instead, they collect traces for a day or two reflecting their behavior once and reuse them.

The contributions of this paper are summarized as follows. First, we show that continuous sensing apps' power use varies significantly depending on user behaviors, which motivates the need for personalized power estimation. Second, we premiere an accurate power estimation system for sensing apps, providing user-specific power estimation at pre-installation time. Third, we present our system optimization for fast estimation and energy-efficient trace collection. Finally, we demonstrate the accuracy and overheads of the system through extensive experiments.

2. MOTIVATION

2.1 Motivating Scenario

Krystal, a businesswoman in her 20s, intends to walk more when she commutes and meets her clients. She searches for a *pedometer* app to organize her walking and is happy to find a long list of apps on Google Play. She carefully selects and installs one with the claim by the developer: "*Consumes less than 5% of battery per day. Tested on a dozen popular phone models.*" However, she gets embarrassed finding her phone unexpectedly powered off in the afternoon. She gets upset, repeating installing and uninstalling one app after another. Positive comments on the battery efficiency were not consistent with her experiences. Even after a week**,** she could not find a satisfactory app. PowerForecaster addresses such discrepancies *in advance*. Figure 1 depicts what she would see for pedometer apps. For each, it shows the estimated decrement of *her own* phone's daily battery life *if she installed and used the app*. She can find the least battery consuming app suitable for her own daily life patterns without relying on doubtful claims and comments from developers and other users.

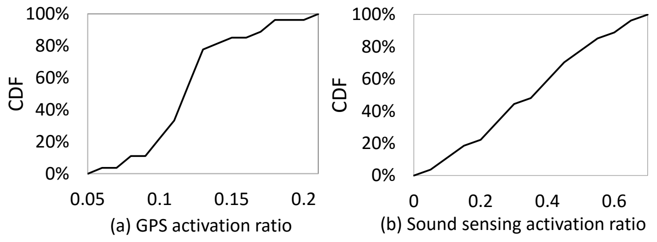


Figure 2. CDF of activation time ratio

2.2 Unveiling the Cause: User Behaviors

What causes such discrepancies between Krystal's experience and the developer's estimates or other users' experiences? It is mainly from diversity in individual user behavior, especially their physical activities and phone use. Users' different activities trigger different branches of logic in the same sensing app. For example, many apps adopt *conditional* sensing pipelines for power optimization, using low and high power sensors selectively [23][30][32]. Actual power use largely depends on how often and long the user's activities trigger the conditions. Different phone use patterns also affect *net power increase of a sensing app* as it *shares* hardware resources with other apps while running in the background [24]. For example, the app obtains already-triggered *wakelocks* at almost zero cost, a major power consumer otherwise.

We quantify the impact of user behavior on battery use with data traces from 27 people over two example sensing apps.

Data traces. We collected sensor and phone usage traces from 27 participants for three weeks (14 undergraduates, 7 company employees in their 20s-50s, 5 graduates in their 20s-30s, and 1 housewife in her 50s). We deployed a data-logging app on their phones. Data collection was set to 12 hours a day (from 10AM to 10PM), but actual collection time varied due to battery depletions. We analyzed data collected more than 8 hours a day.

Applications. We consider two sensing apps inspired by previous works: (1) *MyPath,* a location tracker [41] and (2) *ChatMon,* a conversation monitor [32]. MyPath records the GPS trace of a user every 10 sec. It triggers GPS only when the user is moving which is detected from cheaper accelerometer sensing. ChatMon monitors speakers and conversation turns by sound sensing and processing. Sound sensing is triggered only when the user is close to someone else, which is checked by Bluetooth scans every 2 min.

Effect of physical activities. We first computed the ratio of GPS activation time of MyPath to total execution time. Figure 2(a) shows the cumulative distribution of GPS activation time ratio.

The average ratio per user ranges from 5.7% to 20.2% (mean: 12%, SD: 3.2%). Assuming that MyPath runs 12 hours a day, the top user activates GPS 1.5 hours more than the bottom. We similarly show the ratio of sound sensing activation time of ChatMon, which monitors conversations with the top 5 frequently encountered people. Figure 2(b) shows the cumulative distribution of sound sensing activation time. The average ratio varied from 4.8% to 67.1% (mean: 34.9%, SD: 18.2%). The top user would activate 3.5 more hours of sound sensing than the average user, assuming that ChatMon runs 12 hours a day, which would cause a huge difference in users' power use.

Effect of phone uses. We further examined the effect of resource sharing by analyzing the net increase in CPU activation time caused by wakelocks ChatMon obtained. Here, we briefly discuss two different cases that show such sharing effect. First, two users, P6 and P13, showed similar activation time for sound sensing, but different net increase in CPU activation time. For both users, sound sensing was activated for 29% of the total duration. However, P6's CPU was solely activated by ChatMon for 15% of the total duration, while 22% was activated in case of P13. This difference comes from that P6 used his phone for other purposes for longer duration compared to P13, so P6's CPU was activated more already by other apps. Second, for the other case, P8, P15, and P16 showed similar sound sensing activation time, around 40%. However, they exhibited fairly different net increase in CPU activation time: 30%, 40%, and 33%, respectively. This indicates that phone use of individual users would affect net power consumption of a sensing app due to the sharing effect.

3. POWERFORECASTER DESIGN

3.1 Design Goals

Installation-free: PowerForecaster relieves tedious installation trials caused by sensing apps' unknown battery use. To this end, PowerForecaster needs to provide users with estimated battery consumption before they install and use the apps.

Accuracy: Power estimation should be accurate and reflect what users will experience when they actually use the apps. It needs to be tailored to reflect users' different behaviors.

Latency: Estimation needs to be performed in a short time to support on-the-fly requests from users.

Coverage: PowerForecaster aims at providing power estimation without modifying app binaries; any requirement to change the apps would significantly diminish its utility.

Overhead: PowerForecaster should minimize the overhead on users' mobile phones, especially to pre-collect various sensor traces and device usage traces required for power emulation.

3.2 Power Emulation Approach

We devise a *user behavior-aware power emulation* approach. The key idea is to reproduce the real execution environment of a target app and track its power use while replaying it over pre-collected traces of sensor and device usages. It has three advantages: (1) It accounts for major user-dependent variables, physical activities, and phone use patterns, which significantly affect the target app's power consumption. (2) It estimates power use of an app with no knowledge on its internal logic or prior power profiling required. The emulator tracks hardware use of the app by executing its executable with no need of its source code. (3) It considers shared hardware use with existing apps by reproducing their resource use.

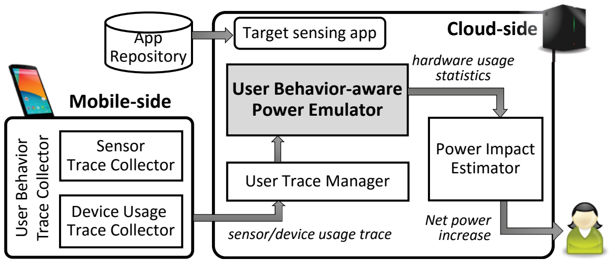

App Repository
Target sensing app
Cloud-side
hardware usage statistics
Mobile-side
User Behavior-aware Power Emulator
User Behavior Trace Collector
Sensor Trace Collector
Power Impact Estimator
User Trace Manager
Device Usage Trace Collector
sensor/device usage trace
Net power increase

Figure 3. PowerForecaster architecture

There have been other approaches to estimate mobile apps' power use, but it is not trivial to apply them to the accurate estimation of sensing apps. *Development-time estimation,* direct measurement with a power meter or model-based schemes [42][56][57], hardly reflects individual user behavior, which largely affects sensing apps' power use. Another potential approach is *collaborative power estimation*. It shares power impact of sensing apps obtained from users who already used the app. The power numbers from the users with similar behaviors are provided to a potential user; a similar concept is proposed in [40] for energy diagnosis. However, this requires a large user base to find similar users due to plenty combinations of behavioral factors affecting power use of sensing apps. We believe PowerForecaster can complement this approach, as it works based on personal traces without a large user pool.

3.3 System Overview

PowerForecaster takes the executable of a target sensing app as an input and provides a personalized estimation of the target app's net power increase (mW) as an output. Later, the power numbers can be processed in various ways for user-friendly presentation (See Section 8 for examples). Figure 3 shows the architecture of PowerForecaster. It consists of two major components, a mobile-side trace collector and a cloud-side power emulator. Prior to power estimation requests, the mobile-side collector pre-collects user behavior traces (possibly for one weekday and one weekend reflecting user behavior). The traces are uploaded to and managed in the cloud server. Upon a request, the power emulator estimates power impact of a target app with the pre-collected traces.

Mobile-side: The mobile-side component collects user behavior traces in the background. The *sensor trace collector* collects a series of sensor data that captures users' physical activities. The *device usage trace collector* logs hardware component usages by existing apps, which are potentially sharable with a new sensing app. Not to interfere with usual phone use, the traces are uploaded to the cloud server only when the phone is on Wi-Fi and charging. The collected traces are used for various sensing apps in the future.

Cloud-side: Upon a power estimation request, the *user behavior-aware power emulator* executes the target app's executable and monitors its hardware usage. At the same time, it replays the sensor and device usage traces to reproduce the execution environments affecting the power impact of the app. As a result, the emulator obtains detailed hardware usage statistics including which, when, and how long hardware components are used. The *power impact estimator* computes the net power increase due to the app based on the cumulative statistics during the emulation.

3.4 System Scope and Limitations

It is extremely challenging to achieve all of our design goals for generic workloads and users. We clarify the scope and limitations of this study and our current system prototype.

Table 1. Sensor data, system call, and events

Sensor Type	Sensor Data	Related System Call	Related System Event
GPS	*longitude, latitude, altitude, speed, bearing*	gps_start(), gps_stop(), …	GPS_EVENT_STARTED, GPS_EVENT_STOPPED,
Bluetooth	*name, address, bond state, type, UUIDs, RSSI*	startDiscoveryNative(), stopDiscoveryNative(), …	ACTION_FOUND, ACTION_DISCOVERY _STARTED/_FINISHED
Wi-Fi	*BSSID, SSID, capabilities, frequency, level*	scan(), wifi_ctrl_recv()	SCAN_RESULTS _AVAILABLE_ACTION
Other sensors (accel. gyro, ...)	*values*	enableSensor(), disableSensor()	

Target apps: Our system focuses on continuous sensing apps that continuously run in the background, such as Google Fit [14], Accupedo [1], and NoomWalk [39]. Specifically, we focus on their autonomous sensing services that perform repetitive sensing tasks controlled by built-in sensing logics and programmatically detectable external events. We do not target conventional apps and sensing apps' UI activities, which are explicitly run and quit by users. The rationale behind this choice is that, for conventional apps, users retain controllability on how much they use the apps and thereby also on the apps' power use to some extent. For sensing apps, continuous sensing is a major drain on power. UI activities consume relatively less power as they are likely run for a short time compared to the whole operation time of sensing apps.

Tracked hardware components: For power estimation, our prototype considers hardware components commonly used by sensing apps: inertial sensors, GPS, Bluetooth and Wi-Fi scans, microphone, and CPU. (See Section 5.3 for details.) We have not yet considered other components such as display, network, or GPU, as those are used less commonly in sensing apps and contribute little to the overall power impact. Section 7 shows that our prototype estimates power use of various sensing apps with average error of 6.6%, considering aforementioned components. We understand that there are a few examples [53] using other components. Our emulation can further expand to consider such components by leveraging various power models proposed in [36][42]. For example, network power cost can be considered by tracking relevant factors such as packet transmission time and signal strength and by reproducing them in the emulator [36].

Selection of user traces: For accurate power estimation, it is important to collect user behavior traces that well reflect a user's common daily behavior. Of course, there could be daily variations of user behaviors, but literature has showed that user behaviors have patterns [8][13][34]. Our ultimate goal is to accurately predict the future power impact of a target app by considering routines and patterns of user behaviors. In this study, we focus on developing a power emulation system that precisely estimates net power increase of an app given proper behavioral traces. We leave capturing a user's routine as a future work.

Scalability to heterogeneous phones: It is well known that it is needed to build power models specific to individual phone models for accurate tracking of apps' power use. This raises a scalability issue, as a variety of phone models exist in today's market. Our system will face the same problem for its wider deployment as the core technique is based on pre-built power models. Note that our prototype incorporates power models for Nexus S and Nexus 5.

To be optimistic, a few phone models take considerable market share currently. Three popular iPhone models and top-10 Android ones take 70%[1] and 33%[2] of iPhone and Android market share, respectively. Power modeling for those popular models will be feasible (even if manual measurements need to be involved). This possibly makes wide deployment of PowerForecaster feasible in practice, dealing with heterogeneous phones. For less-common models, we can consider other approaches. For example, most hardware chipset manufactures open up power profiles per different hardware states, and the profiles can be incorporated into PowerForecaster. In case such information is not available or the power consumption is not stable across the same phone models [6][57], we can leverage prior works to automatically self-construct power models for unknown phones [6][55]; these works achieve an accuracy of 90%-95%. Once power models are available, they can be easily incorporated into our system as we modularize it in a way to include new power models.

Table 2. Representative resource request APIs

Resource type	Request APIs	Parameters
CPU	*acquire*() / *release*() on WakeLock	timeout, ref. count
	setRepeating() / *cancel*() on AlarmManager	alarm type, trigger time, trigger interval
GPS	*requestLocationUpdates*() / *removeLocationUpdates*() on LocationManager	provider, minDelay, minDistance, criteria
Bluetooth	*startDiscovery*() / *stopDiscovery*() on BluetoothAdapter	
Other sensors (Accel., Gyro, ...)	*registerListener*() / *unregisterListener*() on SensorManager	sensor type, sampling rate

4. USER BEHAVIOR DATA COLLECTION

In this section, we explain what data are collected by the trace collector in detail. Section 5 describes how the collected data are used for power emulation.

Sensor trace: The *sensor trace collector* collects sensor data that reflect physical user behaviors. More specifically, it records three types of information. First, it records sensor data streams with timestamps from frequently used target sensors. Second, it records sensor-related system events notified from Android, e.g., *gps_event_started*. Such events are needed to ensure correct replays of sensing apps since many of them use these events as triggers to internal logic. Last, it hooks and logs the system calls and callbacks between the Android framework/kernel and the app. These are required to reproduce power states in sensor devices, as in users' real situations. For example, the time to activate GPS sensing largely depends on whether a user is outdoors or not. Recording GPS values alone is not sufficient to emulate such cold start, which subsequently affects the accuracy of power emulation. Table 1 shows sensor data, events, and system calls that we collect.

Device usage trace: Another important factor for accurate power emulation is to consider shared use of resources, as shown in Section 2.2. A naïve method is to *emulate other apps concurrently with a target sensing app* and accurately trace shared use of various hardware components. However, it requires heavy computation to emulate multiple apps concurrently. It is also challenging to log and replay realistic user inputs for foreground interactive apps. Thus, we take an approach to *record only the device usages of existing apps and estimate sharing effects* on the

[1] http://info.localytics.com/blog/more-iphone-6%E2%80%99s-being-sold-but-iphone-6-shows-stronger-user-engagement

[2] http://info.localytics.com/blog/android-device-samsung-maintains-dominance

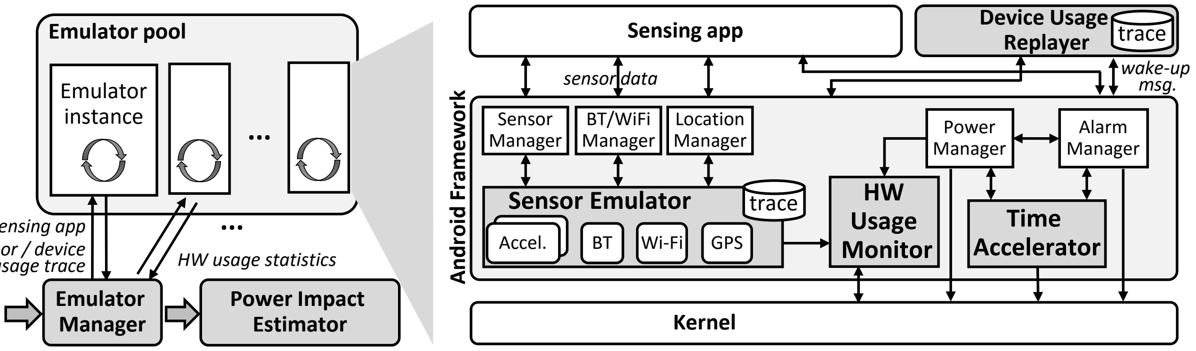


Figure 4. Architecture of user behavior-aware power emulator

emulation process. This enables to accurately account shared use of hardware with other apps without directly executing them.

The *device usage trace collector* records the requests from the existing apps to various hardware components, e.g., wakelocks for CPU, with timestamps. By replaying such requests while emulating the sensing app, it is possible to identify the shared accesses to various hardware components between the target sensing app and other apps. This can be reflected upon accounting the power impact of the target sensing app, resulting in more accurate estimation. Android exposes a narrow set of APIs that allows apps to request access to hardware such as sensors and CPUs. The collector logs a series of the API calls and parameters. Table 2 shows what our prototype collects in more detail.

5. USER BEHAVIOR-AWARE POWER EMULATION

Simply using existing mobile emulators [2][17][54], or their naïve extensions is not a feasible solution to emulate power use of sensing apps. Since their main goal is for testing and debugging of mobile apps, they do not support power monitoring during emulation. Also, they cannot reproduce a user's behavioral environments, which are essential for accurate power estimation. While there are simple sensor simulation tools, their functionality is highly limited for our purpose, as we will discuss in 5.1.

To address the issues, we develop a novel *user behavior-aware power emulator* by significantly extending the existing Android emulator. Figure 4 shows its architecture. Overall operation flow consists of the following three stages.

Pre-emulation stage: Upon a request, the *emulator manager* prepares the necessary inputs for emulation. It first obtains sensor and device usage traces from the *trace manager* as well as the executable of the target sensing app. It then initiates a set of power emulator instances in parallel to reduce the estimation latency (Section 6.1). Each instance installs the target app and receives a part of user traces from the emulator manager. Before executing the app, it replays pre-generated user interaction such as clicking a start button to bootstrap the app if necessary. It also adjusts its system clock to synchronize the time with the traces.

Power-emulation stage: After the preparation, the power emulator instance executes the target sensing app. While the app is running, the *sensor emulator* mimics the operation of sensors to fulfill the app's requests on sensor use by using the pre-collected sensor traces (Section 5.1). To consider the sharing effect, the *device usage replayer* reproduces the existing app's device use based on the device usage trace (Section 5.2). This allows the target app to execute as if it were running along with other apps. During the execution, the *hardware usage monitor* tracks system calls made to use hardware components such as sensors and CPU. At the end, it generates hardware usage statistics, a series of executed system calls along with corresponding timestamps.

Post-emulation stage: With the collected hardware usage statistics, the *power impact estimator* computes the net increase in power consumption by the target app (Section 5.3). For power estimation, we adopt a system call-based method using power profiles obtained by offline profiling [42]. We consider the following hardware components mainly used by sensing apps: *CPU, GPS, inertial sensors, microphone, Bluetooth/Wi-Fi scans*.

5.1 Sensor Emulation

The sensor emulator reproduces a user's physical behavior conditions by feeding the user's real sensor traces to the target app. For realistic emulation, it should feed sensor data at accurate timing and rate as the app demands. It should also emulate the hardware states of the corresponding sensor devices and account for power estimation afterwards.

Existing sensor data replay tools [12][51] do not meet our requirements. SensorSimulator [51] provides a custom library that relays pre-collected sensor data from a host PC to a mobile emulator. However, it requires app-side modification to use the library. ReRan [12] records and replays input events such as touch events and sensor data during the execution of an app for testing and debugging. While it does not require any modification of the app logic, the target app has to be executed during sensor data collection, which violates our pre-installation requirement. Also, both of them do not emulate the power state of sensor devices.

We develop a sensor emulator that mimics real sensor devices' operation and states. First, it provides apps with pre-collected sensor data while fulfilling their specific requests. Second, it replays sensor-related system events for correct app operations. Third, it tracks the changes in expected power states based on the sensor-related system call and callback logs to and from the kernel. The sensor emulator is located between the Android framework and the kernel, which is a middle layer that receives sensor data requests from multiple apps and interacts with the kernel. It hooks the sensor-related system calls from the framework to the kernel and provides sensor data using the collected trace.

Accelerometer, gyroscope, etc: The sensor emulator hooks all sensor activation and deactivation requests from the Android *SensorManager* to the kernel. To handle an activation request, it searches for sensor data corresponding to the request time from the collected trace and pushes them into the sensing app. Also, the sensor emulator performs a sampling of data in the sensor trace if necessary to meet the rate requirement of the sensing app.

GPS: The sensor emulator hooks GPS requests from the Android *LocationManager* to the kernel. Upon a request, it searches for GPS data corresponding to the request time and sends the data after the activation time recorded in the GPS traces. It is important to consider the activation time in a user's real situation, since it varies depending on the user environment (e.g., indoors and outdoors) even with the same request. It affects both the sensing app's execution and the power consumption of the GPS device. At times, there could be no GPS data exactly matching the request time since the GPS request interval of the app can be different from that used for the trace collection. The sensor emulator interpolates GPS data from two adjacent logs and uses them.

Bluetooth and Wi-Fi scan: The sensor emulator intercepts scan requests from the Android *BluetoothAdaptor* and *WifiManager*. Upon the request, it retrieves scan logs with a timestamp closest to the request time from the collected trace and forwards them to the apps. To ensure the apps' proper operation, it broadcasts relevant events, e.g., *bluetooth-discovery-finished*, *scan-results-available-action*, which are recorded in the trace along with scan results.

5.2 Device Usage Replay

The device usage replayer reproduces a user's phone use behavior to reflect the power-sharing effect in the emulation. The replayer runs in the background as an Android service simultaneously with the target sensing app. It uses the pre-collected device usage trace containing a list of time-stamped elements, (*timestamp*, *API function*, *parameter values*) as shown in Table 2. Based on the timestamp, the replayer calls the API with the parameter values.

Since API calls to access hardware components usually operate as a pair, e.g., *acquire()*/*release()*, both API calls should be contained in the trace for the accurate power estimation. However, one API call of the pair could be omitted at times because of duty-cycled data collection (Section 6.2) or the segmentation of device usage traces for parallel execution (Section 6.1). Before replaying the trace, the trace manager fills in the missing calls. For example, if there is only one *release()* in the given trace, it adds *acquire()* and set its timestamp to the beginning of the trace.

5.3 Estimation of Power Impact

During the emulation, the *hardware usage monitor* collects the hardware usage statistics, i.e., a collection of system calls made by the framework, and their timestamps. Sensor-related system calls are captured in the sensor emulator (See Table 1). To account for power use of the CPU, the hardware usage monitor intercepts wakelock requests, *acquire_wake_lock()* and *release_wake_lock()*, from the Android *PowerManager* to the kernel. Upon a request to acquire a wakelock, it also obtains CPU utilization of the target sensing app from */proc/stat* until a release request. We properly scale the CPU utilization to compensate for the different CPU performances between the server and the smartphone as in [36].

After the emulation, the *power impact estimator* computes the net power increase by the target app, $netP_{app}$ based on the hardware usage statistics. We compute $netP_{app}$ as follows:

$$netP_{app} = P^{D}_{with\_app} - P^{D}_{without\_app},$$

where D is the set of hardware components *app* uses and $P^{D}_{with\_app}$ and $P^{D}_{without\_app}$ are the power consumption of D with and without *app*, respectively. As noted earlier, our component set currently supports *CPU, GPS, Bluetooth/Wi-Fi scans, microphone, accelerometer, gyroscope, and magnetometer*.

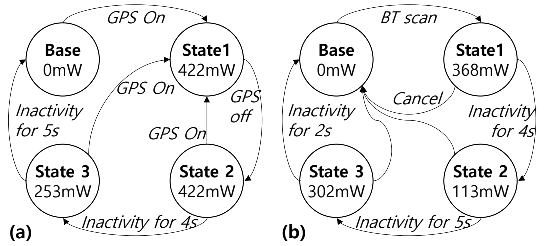


Figure 5. Power model of (a) GPS, (b) Bluetooth on Nexus S

The power consumed by the aforementioned components is estimated with a system call-based power estimation [42]. We built a finite state machine (FSM)-based power model for each component based on associated system calls as in Figure 5. Then, we constructed an FSM for the entire device considering the sharing effect of system calls. The FSM-based power model facilitates to consider tail power state of hardware components and the sharing effect across system calls [42]. To make the power models, we profile the power states of the components for each system call using a power meter [38]. $P^{D}_{with\_app}$ is computed based on FSM models and hardware usage statistics made during the emulation. $P^{D}_{without\_app}$ is done similarly based on the hardware usage statistics after replaying the device usage trace only.

6. SYSTEM OPTIMIZATION

6.1 Acceleration of Emulation

A key challenge of emulation-based power estimation is a long execution time. Unlike typical apps, sensing apps operate closely tied to real-time clocks. They are governed by sampling rates, sensing intervals, and time window for data processing. Naïve emulation would take the same duration of sensor traces to replay. Powerful hardware may not necessarily reduce the emulation time for our workloads, e.g., reading accelerometer at 100Hz for 5 sec.

To address the challenge, we develop 3 acceleration mechanisms: *parallel execution*, *idle time skipping*, and *progressive estimation*. Figure 6 shows them for a commercial pedometer app, Accupedo [1]. We leverage unique characteristics of continuous sensing apps. First, they usually operate by repeating cycles. While the operations could be stateful within a cycle, they might be stateless in between. This implies the potential of parallel emulation. Second, sensing apps wakes up the device as little as possible to save energy; active periods are much shorter than idle periods. Accupedo wakes up every 10 sec to detect the user's movement with 20-ms accelerometer data and sleeps again if no movement. We can safely skip such long idle time to accelerate the emulation.

Parallel execution: Given an original long trace, the *emulator manager* splits it into shorter segments, e.g., 2-min-long each, and assigns a segment to each emulator instance. Each executes a target app independently, replaying sensor and device usage traces in an assigned segment. Upon completion, the hardware usage statistics from each instance are aggregated for total power estimation. A practical issue in parallel execution is to determine a proper length of a segment. It should be short enough to exploit parallelism, but long enough to include the stateful behavior in a cycle. Ideally, proper segment size depends on the apps' internal logic. Consider ChatMon detecting an encounter via BT scan every 2 min. The ideal segment size may be 2 min. Automatically identifying such sizes for each app is not trivial unless we know

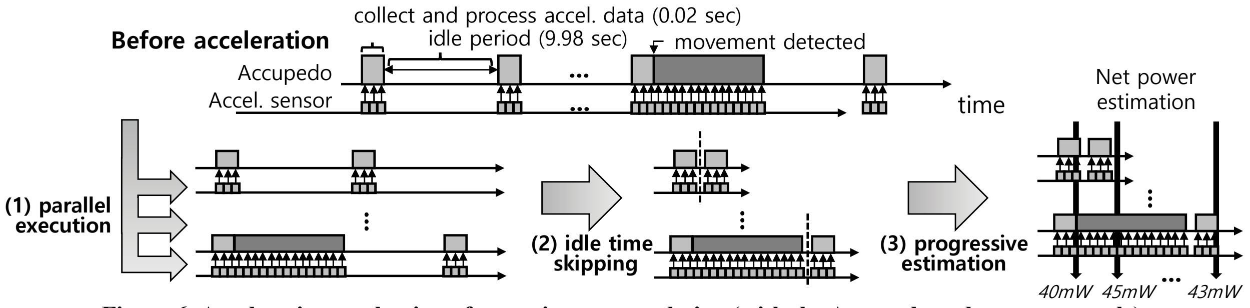


Figure 6. Acceleration mechanisms for sensing app emulation (with the Accupedo pedometer example)

the internal logic of the app in advance. In this paper, we use the same segment size for every app. We found that 2-min-long segments showed reasonable accuracy for all our example apps. An automatic solution might be possible, e.g., by analyzing repeated resource use patterns of a sensing app [33].

Idle time skipping: The *time accelerator* shortens emulation by skipping idle times of the device during the emulation. We regard idle time as a period for which all CPU wakelock requests are released. The time accelerator identifies the start and end of the idle time from the Android *PowerManager*, which manages all wakelocks. Upon detecting all wakelocks have been released (the start time), the time accelerator scans the alarm schedule in the Android *AlarmManager* until the very next alarm (the end time). It skips the idle time by setting the system clock to the end time.

Progressive estimation: Despite the significant reduction of emulation time by the techniques above, the final emulation time is bounded by the segment with the longest execution time. The execution time of a segment varies as idle time skipping is opportunistic upon user behavior. For more responsive service, we develop a progressive estimation. During the emulation, it estimates net power increase with interim results of emulation and progressively updates it. This technique does not reduce the execution time but reduces the first response time to the end user.

6.2 Energy-efficient Trace Collection

The trace collector collects sensor and device usage traces for power estimation of future sensing apps. The trace collection runs only for 1-2 days to reflect user behavior. Still, one might be concerned about the collector's power use, as it is desirable to collect raw sensor data at the highest rate for future generic use.

We applied a widely used duty-cycling technique to reduce power cost of the trace collector. The question is how far can we increase the cycle with minimal decrease of accuracy?. Figure 7 shows the estimation errors with respect to those without duty cycling and respective power overheads for various duty-cycles. We used two 18-hour-long traces of MyPath obtained from the real deployment experiments in Section 8 and set the active data collection duration to 2 min for a period; during the active collection, the sensor data is recorded based on the configuration of a full sensor set (Table 4). Based on the results, we use a 24-min duty-cycle period, i.e., the ratio of 1/12, where the power cost starts to saturate and the error is still below 10% even with the deviation. Such duty cycling is possible since a user's mobility, location and encounter tend to have temporal locality [8][13][34].

7. EVALUATION

We implemented PowerForecaster; on mobile-side, we developed the sensor trace collector as an Android service. We modified the

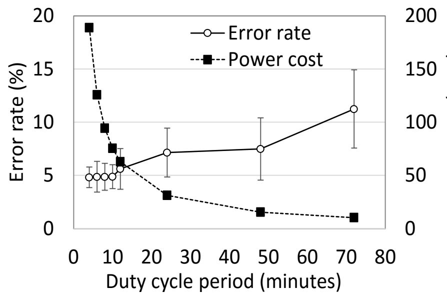


Figure 7. Effect of duty-cycling; bars represent standard deviation

Android system to track device usage and existing apps' hardware usage. On server-side, we developed the power emulator based on the Android emulator. We also made the user trace manager and power impact estimator in Java. PowerForecaster was evaluated for its accuracy, speed, and overhead for power emulation.

7.1 Evaluation Setup

We evaluated PowerForecaster under realistic settings with various apps and users with different behavioral characteristics.

Phones and servers: We used Nexus S (Android 4.1.2) and Nexus 5 (Android 4.4.4) phones for the experiments; we used Nexus S phones by default. For emulation, we arranged 12 desktop servers (with i7-2600k CPU and 16 GB RAM), where each server was configured to run one or more emulator instances in parallel. For a single power estimation request, the average CPU use was 5-10% per each emulator instance. We discuss the server-side resource use in more details in Section 7.3. We did not apply optimization techniques by default.

Sensing apps: We mainly used three sensing apps including a commercial pedometer app, *Accupedo* [1], and the two research apps we developed, *MyPath* and *ChatMon* described in Section 2. Section 7.2.2 describes the applicability of our system to two more commercial apps, *NoomWalk* [39] and *Pedometer2.0* [45]. The hardware components that the three commercial pedometer apps use in common are accelerometers and CPU. Note that, while diverse sensing apps have been proposed in a research domain, only a few types are commercialized, e.g., pedometers. To cover more diverse hardware usage, we selected MyPath and ChatMon. The former uses accelerometers, GPS, and CPU and the latter uses Bluetooth, a microphone, and CPU.

Comparison: For ground truth of the net power increase (mW) of a target sensing app, we used Monsoon power monitors [38]. It is obtained as the difference in power consumptions between when running a target sensing app with existing apps, and when running existing apps only. We made three alternatives that developers can potentially use to provide power impact of sensing apps:

Global-single measures the power using a power monitor while running the target app only, under a specific user behavior scenario, and provides it as an estimation result.

Global-average measures the power uses of the target app under multiple user behavior scenarios. It takes an average value to provide a representative estimation. Note that this does not consider the shared power use with other apps.

Global-single-shared measures the power while running both the target app and apps used in a specific user behavior scenario. Under this scenario, it provides an estimation taking into consideration the shared power use with other apps. Note that this baseline is equivalent to the ground truth of the chosen scenario.

User behavior scenario: For accuracy evaluation, we performed scenario-based experiments. We craft multiple one-hour scenarios with four parameters: mobility, encounter, indoor/outdoor status, and phone usage. We determine the parameter values and their combinations based on real user data described in Section 2. Table 3 summarizes the five scenarios used for our experiments. Figure 8 depicts detailed sequences of user activities and phone usages for *Scenario 4*. Other scenarios are made similarly.

Measurement and trace collection setup: For fair comparison, we need to ensure that the same user behaviors are applied while evaluating all the alternative techniques including the ground truth. We design a setup to easily conduct experiments over various alternatives. Figure 9 shows our setting with four phones (Phone A, B, C, and D) and three power monitors. First, Phone A and B are used for ground truth measurements; Phone A measures power consumption while running a target sensing app with existing apps. Phone B measures power while running existing apps only. The ground truth is calculated by subtracting power measurement of Phone A from that of B. For the PowerForecaster estimation, we configured Phone C to run existing apps and to collect sensor and device usage traces. Phone D measures power while running the target app only. *Global-single(ScenarioID)* and *Global-single-shared(ScenarioID)* use the power measured for a specific scenario while *Global-average* uses the average power measured across the five scenarios. An experimenter drove the cart with the phones and power monitors following each scenario. We used a script to run the same apps (web browser, video player, email client, and map) at the same timings.

7.2 Performance Analysis of PowerForecaster

7.2.1 Accuracy of Net Power Estimation

Figure 11 shows the power measurement with 3 sensing apps in 5 scenarios. Compared to the ground-truth, PowerForecaster accurately estimates net power increases by sensing apps. Average error rate is 6.6%: 5.3% for Accupedo, 6.8% for MyPath, and 7.7% for ChatMon, indicating that PowerForecaster closely traces the power use of the target sensing app. Even for the same app, its power consumptions vary across scenarios due to different user activities and phone usages. For ChatMon, the ground truth of the net power increase ranges from 29 mW to 227 mW.

The three alternatives show much lower accuracy. The average error rate of Global-average is 99% across all scenarios and apps. Global-single(Scenario4) shows an average error rate of 116%, indicating that it misestimates the app's power consumption by more than double compared to the ground truth. The average error rates for Global-single(Scenario2) and Global-single(Scenario3) are 59% and 128%, respectively. This indicates that the approach itself is likely to be error-prone regardless of the scenario since it

Table 3. Summary for five user behavior scenarios

ID	User	Activity	User behavior parameter			
			Moving	Indoor	Encounter	App usage
1	Graduate student (M, 30s)	Shopping alone	50 min	60 min	0 min	5 min
2	Undergraduate student (M, 20s)	Moving and class	10 min	45 min	15 min	20 min
3	Office worker (F, 30s)	Moving and lunch	20 min	40 min	60 min	15 min
4	Office worker (M, 20s)	Going out	30 min	30 min	40 min	30 min
5	Housewife (F, 50s)	Going out	30 min	30 min	40 min	1 min

Time (min)	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58
Mobility		Walking					Staying					Walking										Staying								
In/Outdoor	Outdoor															Indoor														
Encounter	Alone										Encounter																			
Phone usage		Map												Web						Web					Video					

Figure 8. User behavior scenario 4

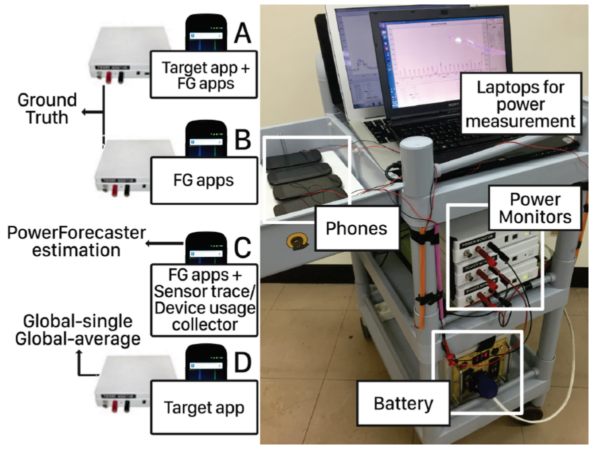

A
Target app + FG apps
Ground Truth
B
FG apps
PowerForecaster estimation
C
FG apps + Sensor trace/ Device usage collector
Global-single Global-average
D
Target app
Laptops for power measurement
Phones
Power Monitors
Battery

Figure 9. Experimental setup

estimates the power use of a target app based on the power measurement for a single specific scenario. Interestingly, even for Scenario 4, Global-single(Scenario4) shows high error rates (86% for Accupedo). This is because Global-single does not reflect the effect of resource sharing with existing app usage. Global-single-shared(Scenario4) which considers this sharing effect, produces an average error rate of 60%. The average error rates for Global-single-shared(Scenario2) and Global-single-shared(Scenario3) are 45% and 83%, respectively. Although the error is 0% for whichever scenario was chosen for this baseline, the average error depends on the differences in behavior modeled in other scenarios.

We analyze the characteristics for each scenario. For Accupedo, Scenario 1 shows the largest net power increase. Its movement duration (50 min) is the longest, consuming much power to process accelerometer data. However, little sharing effect due to short app usage (5 min) makes the largest impact. Scenario 2 is the opposite. Scenarios 4 and 5 reveal the resource sharing effect. They have the same movement behavior but different app usage time, 30 min and 1 min respectively. As a result, scenario 5 shows twice net power increase. MyPath shows a similar trend as it has accelerometer-based triggering. ChatMon exhibits different trends due to different sensing logic and user behavior affecting sensing operation. In scenario 1, there is no BT encounter and thus no audio processing. Accordingly, it shows the smallest net power increase. Scenario 3 has the longest time duration of encounter,

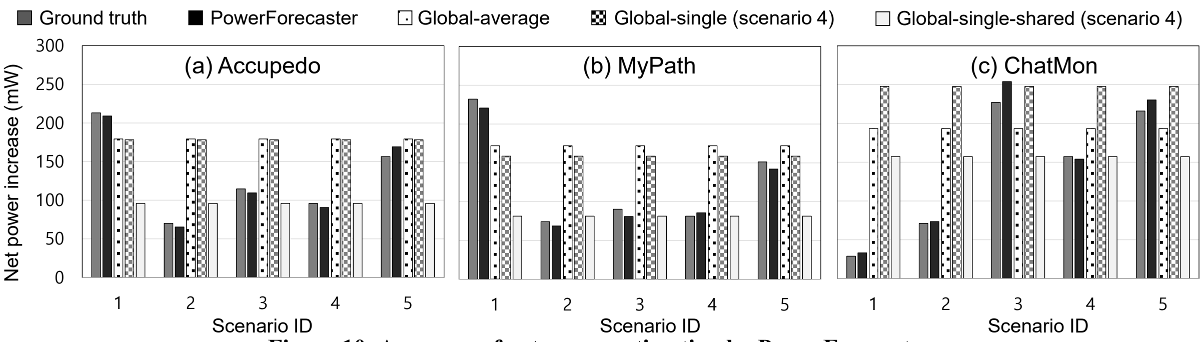


Figure 10. Accuracy of net power estimation by PowerForecaster

60 min., thereby performing sound processing for the entire time. Thus, it shows the largest net power increase. Although it has a greater encounter time of 20 min compared to Scenario 5, its net increase is similar to that of Scenario 5 due to resource sharing.

7.2.2 *Testing with More Apps and Phones*

We show the applicability of PowerForecaster to two more commercial pedometer apps, *NoomWalk* and *Pedometer2.0,* and a more device, Nexus 5. Figure 10(a) shows the results for Scenario 4. Our system accurately estimates the net power increase; the ground truth for NoomWalk and Pedometer2.0 is 9 mW and 103 mW while the estimations are 11 mW and 105 mW, respectively.

Next, we investigate the accuracy of PowerForecaster on Nexus 5 with Android 4.4.4. We built the power model for Nexus 5 and developed the PowerForecaster system based on Android 4.4.4. We conducted experiments with the three sensing apps: Accupedo, MyPath, and ChatMon with Scenario 4, Scenario 1, and Scenario 2, respectively. Figure 10(b) shows that PowerForecaster provides an accurate power estimation even with another phone with a different Android version. Average error rates of PowerForecaster and Global-single are 6% and 21%, respectively.

7.2.3 *Effect of Emulation Acceleration*

We evaluate the emulation acceleration in terms of time-accuracy tradeoff. For parallelism, we use 10, 5, 2, and 1 min segment sizes.

Parallel execution with idle time skipping: Figure 12 shows that the average error rate is still low when applying acceleration; in the figure, w/o skip means that only parallel execution is applied whereas w/ skip means that parallel execution and idle time skipping are all applied. Average error rates are 10.4% and 11.2%, respectively. While they are slightly higher than the those without acceleration, they are still reasonable. As expected, the segment size affects estimation accuracy. Accupedo shows less than or about 10% error rate overall, but the other two apps exhibit relatively large error rates for the 1-min segment. This is due to the characteristics of sensing logic of the apps. We briefly discuss the ChatMon case. For ChatMon, there are two conflicting factors affecting power estimation by parallel execution: the number of BT scans and the delay of sound sensing and processing. With 2-min intervals, ChatMon performs 30 BT scans for 1 hour. In parallel execution, however, the number can vary depending on the segment size. ChatMon is executed independently on each emulator and performs a BT scan when it starts. While the number of scans with 10- and 2-min segment is 30, it increases to 36 and 60 for 5- and 1-min segment, respectively, increasing the estimated power. At the same time, duration for sound processing can decrease in some segments, decreasing in estimated power. Sound processing is triggered only after ChatMon detects a BT

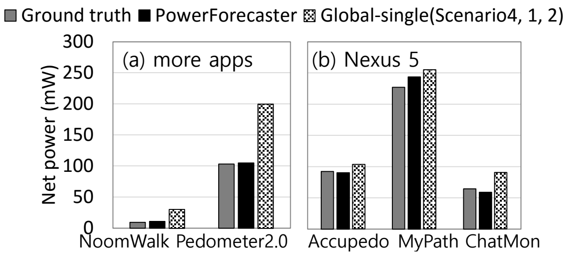


Figure 11. Power measurements with more apps and phones

encounter. Depending on the scenario, the two factors differently contribute to the increase or decrease of estimated power.

Figure 13 shows the CDF of elapsed time to complete emulation of the three sensing apps for all 5 scenarios when applying 2-min segments. For many segments, emulation is finished for less than 20 sec. While about 30% of segments show less than 10 sec of emulation time for Accupedo, the time for 70% of segments is less than 20 sec for MyPath. ChatMon has similar results to Accupedo.

Progressive estimation: Figure 14 shows the average error when applying 2-min segment parallel execution with idle time skipping. The average error quickly decreases within 10 sec for MyPath and Accupedo. MyPath shows 9% of error in 20 sec while Accupedo does 19%. After 30 sec, the error almost saturates. ChatMon takes relatively longer for saturation, i.e., 60 sec. Note that scenario 3 and 4 of ChatMon show saturation, less than 5% in 12 sec. ChatMon's high average error before saturating is attributed to the high error rate in Scenario 1. Its net power increase is very small compared to others, 28.9 mW. Thus, even a small difference in estimation such as 10 mW, results in a large error. An example of such difference is clearly shown in the first 10 sec, where the error rate unexpectedly increases as the estimation progresses. This is mainly due to ChatMon's logic which conducts a BT scan for the first 10 sec upon startup. Progressive estimation regards the power cost of the initial scan as representative of the whole scenario and thus overestimates the results compared to the small ground truth.

We summarize the results for the additional cases introduced in Section 7.2.2. When applying 2-min segment parallel execution with idle time skipping, our system shows 21% error (2 mW difference) for NoomWalk due to its lack of power use and 8% error for Pedometer2.0. For Nexus 5, the error rates are 2%, 5%, and 3% for Accupedo, MyPath, and ChatMon, respectively.

7.3 System Overhead

Mobile-side cost: We examine the mobile-side cost in terms of energy overhead and storage size. We omitted network cost since the trace upload is done only while the phone is on Wi-Fi and charged. We consider two sensor sets, a *basic* set with GPS, accelerometer, and Bluetooth, and a *full* set with all available

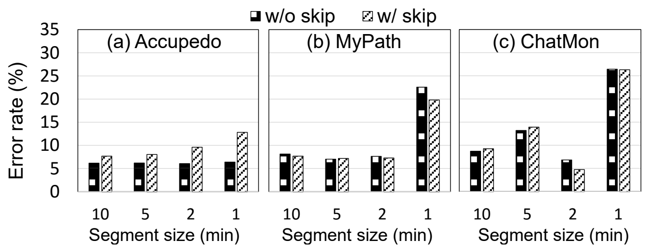
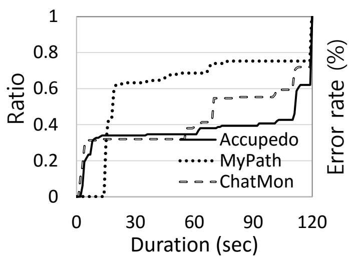


Figure 12. Effect of parallel execution and idle time skipping



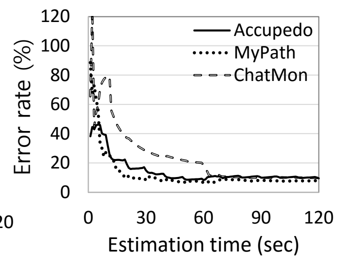


Figure 13. CDF of emulation time (left)
Figure 14. Effect of progressive estimation (right)

sensors. Table 4 shows the results calculated with 1/12 duty cycle, which we set from our empirical experience, as shown in Section 6.2; the expected battery-life decrease is computed assuming 15 hours of battery-life with the fully charged battery, 1500 mAh and 2300 mAh for Nexus S and Nexus 5, respectively. The overall power costs for both basic and full sensor collection were not significant. Even with a full set, the expected lifetime decrement on Nexus S is about an hour; while the cost to collect the sensor data within a cycle is large 390 mW, the average power is reduced to 32 mW due to duty cycling. The cost to collect device usages is negligible in terms of both power use and necessary storage. Note that one hour overhead occurred only a few times since the trace is collected once and used for various different apps.

Server-side cost: The major operations on the server-side are as follows. (1) Prior to a request, the emulator manager stores and manages user behavior traces. (2) Upon a request, it initiates power emulation on distributed phone emulators with the pre-segmented traces. (3) It collects the hardware usage statistics from the emulators and estimates the power impact. The costs of (1) and (3) are not significant. For (1), the network and storage cost is not large, as the maximum data size is about 35 MB for a single day of a user. For (3), the estimation takes less than one sec. For (2), the overhead to run a single emulator instance on our server is about 5% of CPU and 400 MB of memory. The major workload comes from simply emulating a virtual phone image even without foreground apps running; we observed that the extra workload for replaying the trace is quite marginal. We discuss the scalability issue for a worldwide service of PowerForecaster in Section 9.

8. REAL DEPLOYMENT EVALUATION

Experiment setup: We recruited 6 graduate students and 1 researcher(P1) on campus in Nov. 2014. We provided a Nexus S running our trace collector and had them use it as their primary phone. They installed their own SIM cards on the phone and the apps they usually use. Not to affect their usual phone use and corresponding power consumption, we provided another Nexus S to collect sensor data at the same time. To ensure the same user behaviors and environmental conditions for both phones, we taped them up to be carried together. For comprehensive analysis, we collected the full sensor data from 8 a.m. to 2 a.m. the next day.

Table 4 Mobile-side cost; accelerometer/light/proximity/magnetic (delay_fastest), gyroscope(delay_game), GPS/BT/3G/WiFi(1 min.)

Data	Storage (MB)	Avg. power (active period power) (mW),		Expected battery-life decrease (h)	
		Nexus S	Nexus 5	Nexus S	Nexus 5
device usage	0.24	< 1 (<1)	< 1 (<1)	-	-
sensor(basic)	11.1	23 (276)	25 (298)	0.8	0.5
sensor(full)	34.6	32 (390)	26 (314)	1.1	0.6

Each participant was compensated KRW 200,000 (USD 179). The experiments were conducted for two weeks. The first week was to collect prerequisite information to estimate power impact, including sensor traces, device usage traces, hardware usage statistics, and battery levels. When the second week started, each participant installed one of three sensing apps, Accupedo, MyPath, or ChatMon. The second week was to measure the decrease of the battery life when the participants used the designated app. For ChatMon, we recruited two students from the same project group and configured their ChatMons to detect each other.

User-friendly power impact estimation: For this study, we further process *netP_{app}* in more user-friendly way. Specifically, we convert *netP_{app}* to the expected decrease in the battery life (in hours) of the user's phone, by applying the following formula:

$$decrease(app) = battery\text{-}life_{without\_app} - battery\text{-}life_{with\_app}$$
$$= \frac{capacity}{P_{without\_app}} - \frac{capacity}{(P_{without\_app} + netP_{app})},$$

where *capacity* is the phone's full battery capacity and $P_{without\_app}$ is the average power use of the phone before running the target sensing app. $netP_{app}$ is the net power increase of the app outputted by our emulator. For the user-friendly output, our collector logs phones' battery levels to estimate $battery\text{-}life_{without\_app}$. We use a simple method to calculate $battery\text{-}life_{without\_app}$: the reciprocal of the battery drain rate (%/h), computed by using the consecutive samples of <timestamp, battery level> as in [10][40].

Results: Table 5 shows the battery-life decrease of each sensing app for the 7 participants. The sensing apps reduced the phones' battery life by 12.1 hours on average. Interestingly, even a commercial app, Accupedo reduces battery life by 5.3-14.7 hours.

We investigate the estimation accuracy. We asked the participants to select two days during the first week, each with different IDs, on Table 6. Note that it is impossible to measure exact estimation accuracy without complete regeneration of a day's user behavior. Instead, we indirectly compare the estimated battery life with average battery life during the second week. For each selected day, Table 6 shows PowerForecaster's observed battery life and estimated future life, taking the user behavior on that day into the input. It also shows the actual second week battery life when the user used the sensing app. PowerForecaster estimates battery-life decreases with high accuracy even in uncontrolled real-life settings, over 90% for 10 cases out of 14. Even for Accupedo, a commercial app, accuracies were over 90% for all six cases.

There are few cases when the estimation was not accurate, e.g., for P5 with the trace 5-4. P5 mentioned he did not use the phone as usual, nor did he move much since it was a weekend day. P5's battery life was about 20 hours on average during the first week, but 33 hours in trace 5-4. This observation leads us to separately estimate power behaviors on weekends. We will extend PowerForecaster to make progressive classification of user's daily life patterns.

Table 5. Summary of real-deployment experiment

ID	Age	2nd-week sensing app	1st-week avg. battery life (h)	2nd-week avg. battery life (h)	Avg. battery life decrease (h)
P1	37	Accupedo	26.6	21.3	5.3
P2	26		49.6	34.8	14.7
P3	32		30.5	17.5	13.0
P4	32	MyPath	30.2	19.7	10.5
P5	23		19.0	15.3	3.7
P6	33	ChatMon	29.1	16.6	12.5
P7	24		39.3	14.8	24.5

Table 6. Estimation of battery life

ID	TraceID	Battery life (h)	Estimated (future) battery life (h)	2nd-week avg. battery life (h) / stdev
P1	1-4	24.5	**21.1**	**21.3** / 0.9
	1-6	26.2	**22.7**	
P2	2-5	50.0	**37.0**	**34.8** / 2.4
	2-7	47.5	**35.7**	
P3	3-4	24.6	**17.7**	**17.5** / 2.9
	3-7	22.4	**17.2**	
P4	4-1	33.2	**24.1**	**19.7** / 2.7
	4-7	24.8	**20.0**	
P5	5-4	33.6	**24.1**	**15.3** / 2.6
	5-5	17.7	**15.3**	
P6	6-1	24.5	**15.2**	**16.6** / 1.9
	6-5	45.6	**18.1**	
P7	7-2	41.3	**18.0**	**14.8** / 1.8
	7-3	42.7	**19.6**	

We also examine the estimation accuracy when applying all the optimization techniques. For 1/12 duty cycling, we picked out 2-min data every 24-min from sensor and device usage traces. We performed power emulation using the sampled traces with 2-min segment parallel execution and idle time skipping. Figure 15 shows the progressive estimation errors over time with trace 4-1 and 7-3, compared to the case using the whole trace without any acceleration techniques. Notably, with only 1/12 of the trace, PowerForecaster shows only 8.5% and 3.4% of errors, respectively. This is because the sampled data still represent the rest due to the context continuity. For the trace 4-1, our system reveals decreasing errors over time, 13% and 10% at 20 and 30 sec, respectively. For trace 7-3, it saturates to 3% after 5 sec.

Besides battery-life impact, PowerForecaster also provides in-depth analysis on future power behavior of sensing apps by using contextual information obtained from the sensor trace. Figure 16 shows the estimated power hotspots that P4 saw prior to use the MyPath app depending on the time and the location, respectively. While PowerForecaster basically targets end users, it can also be used to support developers to estimate battery-life impact of their apps in real-life situations. Based on behavior traces collected from diverse users, our system can provide the estimated impact of their apps over different users before the real deployment.

9. DISCUSSION

Daily variation of user behaviors: We focus on providing an accurate power impact estimate of a sensing app given users' behavioral traces of a specific day. However, it is arguable that the estimation for a past specific day can well represent the real power impact in the future. We believe that there are both possibilities and limitations here. The deployment study result in Section 8 shows that the estimation accuracy is reasonably good, which implies that there are likely to be similar behavioral patterns over

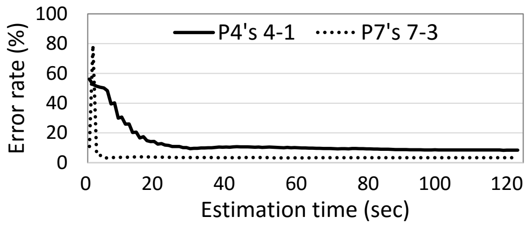


Figure 15 End-to-end performance with P4's 4-1 and P7's 7-3

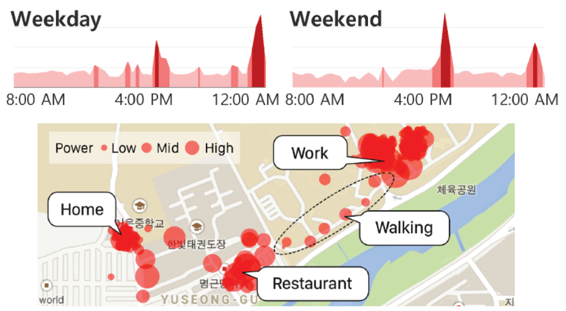


Figure 16 Expected power behavior of MyPath for P4

days. At the same time, there are also cases that exhibit large deviation from the estimate depending on users. From the results, it might be difficult to provide a single representative estimate of power impact based on one-day-long traces in the face of daily variations of behaviors. To address the problem, we can consider collecting user traces for several days (possibly periodically) and making an estimate in a more detailed form, e.g., a reasonable range instead of a single value or separate estimates for weekdays and weekends. While this can increase the mobile side cost, e.g., about an hour or less decrease in daily battery life for several days or a week, it might be a useful option for users who do not mind collecting traces for a longer period of time. Also, if it is possible, it may be able to recognize the behavioral patterns and model the user behaviors as studied in [8][13][34]. Such a model may allow us to estimate power impact of sensing apps more systematically.

Scalability: To complete a power forecasting request for a given user and a given sensing app, it takes about 30 seconds on average with a total of 45 parallel emulator instances running on two physical servers of the specifications aforementioned in Section 7.1. For the cost estimation, we used MyPath and an 18-hour-long trace from user P4 (see Section 8) under the optimization settings of 1/12 duty cycle and 2-min-long segmentation. To discuss the scalability for worldwide deployment, we attempt to estimate the volume of cloud infrastructure partly based on publicly available data and partly with our reasonable assumptions where no public data is available to the best of our knowledge. According to Google I/O 2015 [15], there are 50 billion app installations per year from 1 billion Android users. We do not know the number of sensing app installations out of those; we attempt to assume that 1 billion ones are for sensing apps. This guess is an aggressively high number based on (1) the fact that Google Fit, one of the most popular sensing apps, has been cumulatively installed about 5 million times (0.01% of 50 billion), and (2) the prospect that sensing apps will proliferate in the near future. Given these numbers, we expect about 2.8 million sensing app installations per day. The number of daily power forecasting requests may be higher than this; users may want to compare a few similar apps before installing one. We use 3 for a reasonable number of comparisons; it produces a total of 4 power forecasting requests, giving 11.2 million power forecasting requests per day worldwide.

Given the resources to complete a single request shown above, this worldwide workload could be handled by 7800 physical servers[3]. This would be reasonably practical, based on the existing server volume of today's commercial cloud such as 1 million servers in Microsoft datacenters[4].

Our estimation has a number of limitations. The true computing powers of a commercial cloud server are unknown. It lacks the notion of diurnal workload fluctuations, which may act adversely to the estimated server volume. The ever-growing popularity of sensing apps would also aggravate the scalability issue. On the contrary, a number of advanced strategies may be able to resolve the scalability issues. For example, (1) we can make the kernel ticks advance faster by modifying the Android emulator's kernel timer; it will further shorten the time to complete the processing of a given trace. (2) An emulator instance may be able to accommodate multiple target apps; this would be able to save a significant amount of computing resources as the major workload comes from emulating a phone image, rather than from running apps on top of it. (3) The service provider may predict an installation of a particular sensing app based on popularity or user profiles; making an estimation in advance by temporary surplus cloud resources would distribute the server-side peak loads.

Power estimation of advanced techniques for sensing apps: For power estimation of sensing apps, we currently focus on their repetitive sensing and subsequent data processing controlled by built-in logics. However, some apps could offload their heavy computation logics on the server to save energy [47][48]. To address such apps, it is required to track their network usage and estimate the corresponding power cost. We will extend our system to support power estimation of sensing apps' network usage. As in Section 3.4, a possible method would be to track network-relevant factors on a user's phone and reproduce them in the emulator [36]. Besides the cloud offloading, low power processors have been used for energy-efficient sensor data processing. Our system can be extended to handle such advanced architectures by building power models for low power processors and tracking their usages.

Privacy: To benefit from PowerForecaster, users are required to upload their sensor and device usage data into the cloud, which naturally raises privacy concerns. Optimistically, such concerns may be mitigated if a system provides sufficient utility [26]. Many users already upload their privacy-sensitive data such as location and photos to the cloud for better management and accessibility. However, to be conservative, privacy is a very subjective and sensitive issue [18]. To relieve the concern, our system will adopt proper solutions, e.g., allowing users to control the data type and granularity to upload as well as data collection period.

10. RELATED WORK

Continuous sensing apps and energy optimization: Recently, lots of continuous sensing apps have been proposed in a research domain [30][32][37][52]. In-depth exploration can be found in [25]. While much effort has been put toward developing an accurate recognition logic, they also adopt diverse optimization techniques to save battery. Also, many works propose common platforms for mobile sensing apps and system support for energy-efficient context sensing [5][11][19][20][21][22][29][31][46].

Power profiling and modeling: Power profiles and models are important baselines for energy optimization in mobile systems. There have been extensive efforts to build accurate models for the power consumption of mobile apps and phone H/W [6][7][36][42][43][55][56][57], on which the power model that we adopt for accurate power estimation is built. Similar to our emulation-based approach, WattsOn provides a tool to emulate apps' power use in development environments to support app developers [36]. Unlike our work, it targets interactive foreground apps and thus focuses on power emulation for display, CPU, and network. We, however, address the power impact of background sensing apps.

Energy diagnosis of running apps: Several works address energy diagnosis problems such as abnormal battery drain due to bugs and misconfigured apps [4][33][40][44]. They help users find the causes of undesirable battery drain and fix them. We complement these works with the expected power impact of continuous sensing apps before installing them, helping users make informed decision in advance.

Human battery interaction: Literatures on human-battery interaction for mobile users examined battery-charging behavior [9][49], user perception of battery interface [10][49], and change of user behavior by battery awareness [3]. However, since they mostly focus on conventional mobile apps, they did not address issues and concerns related to continuous sensing apps. As an early attempt, our previous study reported users' concerns about running continuous sensing apps [35]. Their different battery drain depending on user behavior could embarrass users due to disparities between users' anticipation of the near-future battery status and the actual outcome. It proposed a novel tool to provide user behavior-dependent battery drain information while running sensing apps, and showed its effectiveness. Unlike this work, PowerForecaster focuses on providing power impact of sensing apps at pre-installation time and thereby helps users make better informed decision in advance.

Mobile app testing: Recently, there have been research efforts to develop autonomous testing frameworks for mobile apps using mobile emulators [16][50][27]. They use a monkey tool to automate the execution of mobile apps by generating streams of user interface events and analyze runtime properties such as app crashes and page contents systematically on the emulator. Our system differs from those in two aspects. First, for the execution of sensing apps, we replay sensor data streams and sensor status, which are the major input of sensing apps. Second, we emulate the power state of sensor devices for the power impact estimation.

11. CONCLUSION

In this paper, we present *PowerForecaster*, a system that provides users with personalized power impact of continuous sensing apps prior to installation. We show that individual user behavior is a key to understand power impact of continuous sensing apps. We developed a novel user behavior-aware power emulator estimating power use by sensing apps based on user's real behavioral traces. We implemented and extensively evaluated the PowerForecaster prototype in terms of accuracy, speed, and system overhead.

12. ACKNOWLEDGMENTS

We thank our shepherd, Dr. Kiran Rachuri and anonymous reviewers for their valuable comments. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2011-0018120). The corresponding author is Seungwoo Kang.

[3] 2 servers × 0.5 min. × 5.6e+6 requests = 5.6e+6 server•min. = 3888 server•day.
[4] http://www.microsoft.com/en-us/server-cloud/cloud-os/global-datacenters.aspx

13. REFERENCES

[1] Accupedo. http://play.google.com/store/apps/details?id=com.corusen.accupedo.te

[2] Android emulator. http://developer.android.com/tools/help/emulator.html

[3] Athukorala, K., Lagerspetz, E., von Kügelgen, M., Jylhä, A., Oliner, A. J., Tarkoma, S., and Jacucci, G. How carat affects user behavior: implications for mobile battery awareness applications. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems* (CHI), 2014.

[4] Battery Doctor. http://ksmobile.com/product/battery-doctor.html

[5] Chu, D., Lane, N. D., Lai, T. T. T., Pang, C., Meng, X., Guo, Q., Li, F., and Zhao, F. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems* (SenSys), 2011.

[6] Dong, M., and Zhong, L. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (MobiSys), 2011.

[7] Dong, M., Lan, T., and Zhong, L. Rethink energy accounting with cooperative game theory. In *Proceedings of the 20th annual international conference on Mobile computing and networking* (MobiCom), 2014.

[8] Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., and Estrin, D. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (MobiSys), 2010.

[9] Ferreira, D., Dey, A. K., and Kostakos, V. Understanding human-smartphone concerns: a study of battery life. In *Pervasive Computing*, 2011.

[10] Ferreira, D., Ferreira, E., Goncalves, J., Kostakos, V., and Dey, A. K. Revisiting human-battery interaction with an interactive battery interface. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing* (UbiComp), 2013..

[11] Georgiev, P., Lane, N. D., Rachuri, K. K., and Mascolo, C. DSP.Ear: Leveraging CO-Processor Support for Continuous Audio Sensing on Smartphones, In *Proceedings of the 12th ACM Conference on Embedded Networked Sensor Systems* (SenSys), 2014.

[12] Gomez, L., Neamtiu, I., Azim, T., and Millstein, T. Reran: Timing-and touch-sensitive record and replay for android. In *Proceedings of 35th International Conference on Software Engineering 2013* (ICSE), 2013.

[13] Gonzalez, M. C., Hidalgo, C. A., and Barabasi, A. L. (2008). Understanding individual human mobility patterns. *Nature*, *453*(7196), 779-782.

[14] Google Fit, https://play.google.com/store/apps/details?id=com.google.android.apps.fitness

[15] Google I/O 2015, https://events.google.com/io2015

[16] Hao, S., Liu, B., Nath, S., Halfond, W. G. J., and Govindan, R. PUMA: Programmable UI-Automation for Large-Scale Dyanmic Analysis of Mobile Apps. In *Proceedings of the 12th international conference on Mobile systems, applications, and services (MobiSys)*, 2014.

[17] iOS simulator. https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/iOS_Simulator_Guide/

[18] Itani, W., Kayssi, A., and Chehab, A. Privacy as a Service: Privacy-Aware Data Storage and Processing in Cloud Computing Architecture. In *Proceedings of the 8th IEEE International Conference on Dependable, Autonomic and Secure Computing* (DASC), 2009.

[19] Ju, Y., Lee, Y., Yu, J., Min, C., Shin, I., and Song, J. SymPhoney: a Coordinated Sensing Flow Execution Engine for Concurrnet Mobile Sensing Applications. In *Proceedings of the 10th ACM Conference on Embedded Networked Sensor Systems* (SenSys), 2012.

[20] Ju, Y., Min, C., Lee, Y., Yu, J., and Song, J. An Efficient Dataflow Execution Method for Mobile Context Monitoring Applications. In *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications* (PerCom), 2013

[21] Kang, S., Lee, J., Jang, H., Lee, Y., Park, S., and Song, J. (2010). A scalable and energy-efficient context monitoring framework for mobile personal sensor networks. *IEEE Transactions on Mobile Computing*, *9*(5), 686-702.

[22] Kang, S., Lee, Y., Min, C., Ju, Y., Park, T., Lee, J., Rhee, Y., and Song, J. Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *Proceedings of the 8th IEEE International Conference on Pervasive Computing and Communications* (PerCom), 2010

[23] Kim, D. H., Kim, Y., Estrin, D., and Srivastava, M. B. Sensloc: sensing everyday places and paths using less energy. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems* (SenSys), 2010.

[24] Lane, N. D., Chon, Y., Zhou, L., Zhang, Y., Li, F., Kim, D., Ding, G., Zhao, F., and Cha, H. Piggyback CrowdSensing (PCS): energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (SenSys), 2013.

[25] Lane, N. D., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., and Campbell, A. T. A survey of mobile phone sensing. *Communications Magazine, IEEE*, *48*(9), 140-150.

[26] Lee, J. and Hoh, B. Sell Your Experiences: A Market Mechanism based Incentive for Participatory Sensing. In *Proceedings of the 8th IEEE International Conference on Pervasive Computing and Communications* (PerCom), 2010.

[27] Lee, K., Flinn, J., Giuli, T. J., Noble, B., and Peplin, C. AMC: Verifying User Interface Properties for Vehicular Applications. In *Proceedings of the 11th international conference on Mobile systems, applications, and services (MobiSys)*, 2013.

[28] Lee, Y., Iyengar, S. S., Min, C., Ju, Y., Kang, S., Park, T., Lee, J., Rhee, Y., and Song, J. (2012). Mobicon: a mobile context-monitoring platform. *Communications of the ACM*, 55(3), 54-65.

[29] Lee, Y., Ju, Y., Min, C., Kang, S., Hwang, I., and Song, J. CoMon: Cooperative Ambience Monitoring Platform with Continuity and Benefit Awareness. In *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys)*, 2012.

[30] Lee, Y., Min, C., Hwang, C., Lee, J., Hwang, I., Ju, Y., Yoo, C., Moon, M., Lee, U. and Song, J. Sociophone: Everyday

face-to-face interaction monitoring platform using multi-phone sensor fusion. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services* (MobiSys), 2013.

[31] Lu, H., Yang, J., Liu, Z., Lane, N. D., Choudhury, T., and Campbell, A. T. The Jigsaw continuous sensing engine for mobile phone applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems* (SenSys), 2010.

[32] Luo, C., and Chan, M. C. SocialWeaver: collaborative inference of human conversation networks using smartphones. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (SenSys), 2013.

[33] Ma, X., Huang, P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L.K., and Voelker, G. M. eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In *Proceedings of 10th USENIX Symposium on Networked Systems Design and Implementaton* (NSDI), 2013.

[34] Miklas, A. G., Gollu, K. K., Chan, K. K., Saroiu, S., Gummadi, K. P., and De Lara, E. Exploiting social interactions in mobile systems. In *Proceedings of the 9th international conference on ubiquitous computing* (UbiComp), 2007.

[35] Min, C., Yoo, C., Hwang, I., Kang, S., Lee, Y., Lee, S., Park, P., Lee, C., Choi, S., and Song, J. Sandra Helps You Learn: the More You Walk, the More Battery Your Phone Drains. In *Proceedings of the 2015 ACM international joint conference on Pervasive and ubiquitous computing* (UbiComp), 2015.

[36] Mittal, R., Kansal, A., and Chandra, R. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking* (MobiCom), 2012.

[37] Mohan, P., Padmanabhan, V. N., and Ramjee, R. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems* (SenSys), 2008.

[38] Monsoon power monitor. http://msoon.com/LabEquipment/PowerMonitor/

[39] NoomWalk. http://play.google.com/store/apps/details?id=com.noom.walk

[40] Oliner, A. J., Iyer, A. P., Stoica, I., Lagerspetz, E., and Tarkoma, S. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (SenSys), 2013.

[41] Paek, J., Kim, J., and Govindan, R. Energy-efficient rate-adaptive GPS-based positioning for smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (MobiSys), 2010.

[42] Pathak, A., Hu, Y. C., and Zhang, M. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems* (EuroSys), 2012.

[43] Pathak, A., Hu, Y. C., Zhang, M., Bahl, P., and Wang, Y. M. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems* (EuroSys), 2011.

[44] Pathak, A., Jindal, A., Hu, Y. C., and Midkiff, S. P. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (MobiSys), 2012.

[45] Pedometer2.0, http://play.google.com/store/apps/details?id=com.csero.pedometer

[46] Ra, M., Priyantha, B., Kansal, A., and Liu, J. Improving Energy Efficiency of Personal Sensing Applications with Heterogeneous Multi-Processors. In *Proceedings of the 2012 ACM international joint conference on Pervasive and ubiquitous computing* (UbiComp), 2012.

[47] Rachuri, K. K., Efstratiou, C., Leontiadis, I., Mascolo, C., and Rentfrow, P. J. METIS: Exploring mobile phone sensing offloading for efficiently supporting social sensing applications. In *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications* (PerCom), 2013.

[48] Rachuri, K. K., Mascolo, C., Musolesi, M., and Rentfrow P. J. SociableSense: Exploring the Trade-offs of Adaptive Sampling and Computation Offloading for Social Sensing. In *Proceeding of the 17th ACM Conference on Mobile Computing and Networking* (MobiCom), 2011.

[49] Rahmati, A., and Zhong, L. (2009). Human–battery interaction on mobile phones. *Pervasive and Mobile Computing*, *5*(5), 465-477.

[50] Ravindranath, L., Nath, S., Padhye, J., and Balakrishnan, H. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th international conference on Mobile systems, applications, and services (MobiSys)*, 2014.

[51] Sensor Simulator, https://code.google.com/p/openintents/wiki/SensorSimulator

[52] Tan, W. T., Baker, M., Lee, B., and Samadani, R. The sound of silence. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (SenSys), 2013.

[53] Tian, H., Ruogu Z., and Guoliang Xing. COBRA: color barcode streaming for smartphone systems. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (MobiSys), 2010.

[54] Windows phone 8.1 emulators. http://www.microsoft.com/en-us/download/details.aspx?id=44574

[55] Xu, F., Liu, Y., Li, Q., and Zhang, Y. V-edge: Fast Self-constructive Power Modeling of Smartphones Based on Battery Voltage Dynamics. In *Proceedings of the 10th USENIX SymPosium on Networked Systems Design and Implementation* (NSDI), 2013.

[56] Yoon, C., Kim, D., Jung, W., Kang, C., and Cha, H. AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring. In *USENIX Annual Technical Conference,* 2012.

[57] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2010.