Scalable Power Impact Prediction of Mobile Sensing Applications at Pre-Installation Time

Chulhong Min, Youngki Lee, Chungkuk Yoo, Inseok Hwang, Younghyun Ju, Junehwa Song, and Seungwoo Kang, *Member, IEEE*

Abstract—Today's smartphone application (hereinafter 'app') markets do not provide information on power consumption of apps, which is essential for users. Continuous sensing apps make this problem more severe because significant power is consumed without the users' awareness. We propose *PowerForecaster* to break through such an exhaustive cycle. It provides users with personalized estimation of sensing apps' power cost at pre-installation time. It is challenging to provide such estimation in advance because the actual power cost of a sensing app varies depending on user behavior such as physical activities and phone use patterns. To address this, we develop a novel *power emulator* as a core component of PowerForecaster. It achieves accurate, personalized power estimation by reproducing users' behaviors and emulating the target app's power use. We optimize the system to make the power emulation fast and its trace collection energy efficient. We further address the problem of dealing with large-scale emulation requests from worldwide deployment. We develop a novel selective emulation approach to minimize the server-side resource cost. We performed extensive experiments and the experimental results show that PowerForecaster achieves the power estimation accuracy of 93.4 percent and saves on 60 percent of the emulator instance usage.

Index Terms—Energy management, mobile applications, smart devices, system architecture

1 Introduction

Users select desirable mobile apps by considering diverse information provided by smartphone app markets, such as app features, screenshots, and user comments. However, a key piece of information is still missing, which is *power consumption* by an app. Users can determine the app's power cost empirically only after they install and use it for a long period of time. Based on such experiential perception, users make a decision whether they keep using the app or not. However, such experiential power control is no longer effective for continuous sensing apps [1], [2]. Users cannot be explicitly aware of those apps' power cost as the apps consume the power continuously in the background. A pedometer app, Accupedo [3], for example, uses up to 200 mW of power additionally depending on situations and could cause an early shutdown of the smartphone.

What if users could see the estimated power cost of a sensing app on the app market prior to its installation? If so, the app market could relieve users of a significant burden from exhaustive trial and error and help them make a judicious decision to choose a certain app. Also, if users accept the expected power cost of the selected app, they might not be bothered even when the battery is rapidly running low.

Realizing such a function for continuous sensing apps is not straightforward. A simple way is that app developers share the average power cost of the apps for common use cases. However, we uncovered that such a developer-driven report could be inaccurate for many individual users. The main reason is that the power cost of a sensing app is largely different depending on users' behaviors such as physical activities, phone use, and other environmental factors [4], [5], [6], [7]. The error could increase when various optimization techniques are applied to the sensing app, e.g., activating power-hungry GPS based on energy-efficient IMU.

In this paper, we present *PowerForecaster*. It provides an *instant*, *personalized* power cost of a sensing app *prior to the installation*, capable of being integrated into mobile app markets. Fig. 1 depicts the mockup screenshots of services that would be enabled by PowerForecaster. Our system provides a number of unique user experiences. First, PowerForecaster offers the power impact of sensing apps at *pre-installation time*. By doing so, it removes users' hassle of installing, using, and uninstalling apps one after another. They usually repeat this process until they find energy-efficient apps. Second, PowerForecaster estimates highly *personalized* power use by reflecting an individual user's patterns of activity and phone

- C. Min is with the Nokia Bell Labs, Cambridge CB3 0FA, United Kingdom. E-mail: chulhong.min@nokia-bell-labs.com.
- Y. Lee is with the Department of Computer Science and Engineering, Seoul National University, Gwanak-gu, Seoul 08826, Republic of Korea. E-mail: youngkilee@snu.ac.kr.
- C. Yoo and I. Hwang are with IBM Research, Austin, TX 78758. E-mail: ckyoo@ibm.com, ihwang@us.ibm.com.
- Y. Ju is with the Hyundai Motor Company, Seocho-gu, Seoul 06797, Republic of Korea. E-mail: younghyun.ju@hyundai.com.
- J. Song is with the School of Computing, KAIST, Daejeon 34141, Republic of Korea. E-mail: junesong@nclab.kaist.ac.kr.
- S. Kang is with the School of Computer Science and Engineering, KOREATECH, Cheonan-si, Chungcheongnam-do 31253, Republic of Korea. E-mail: swkang@koreatech.ac.kr.

Manuscript received 11 Apr. 2018; revised 5 Mar. 2019; accepted 21 Mar. 2019. Date of publication 9 Apr. 2019; date of current version 5 May 2020. (Corresponding author: Seungwoo Kang.) Digital Object Identifier no. 10.1109/TMC.2019.2909897

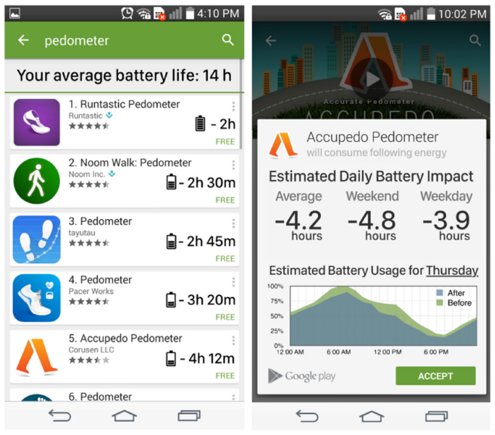


Fig. 1. Power impact at pre-installation time.

use. It ensures higher accuracy of power estimation. Third, a wide variety of sensing apps are supported without requiring any modification of the app logic nor requiring further information from developers. Last, it achieves *scalable* processing to support a worldwide deployment.

To enable the aforementioned unique features, we propose a *trace-driven emulation* approach. First, it pre-collects sensor and device usage traces from a user's phone. The traces represent user's physical activities and phone use patterns, respectively. Second, it estimates the expected power use of a target app based on a *user behavior-aware power emulator* with the collected traces. The target app is executed on the emulator by replaying the sensor trace. While executing the app, the emulator tracks changes in the power states of hardware components used by the app. At the same time, the hardware use of other apps is also tracked by replaying the device usage trace. This is to accurately estimate net power increase by the target app. A sensing app often shares hardware with other apps as it runs in the background and it should be accounted for properly. After the emulation, each power state by hardware use is converted into a power number and total power consumption is obtained by aggregating the power numbers. In this way, PowerForecaster achieves the accurate estimation of the power impact of a sensing app, by considering user behavior in the power emulation.

To make our approach feasible, we address a number of technical challenges. Most importantly, we design our own Android *power emulator* with significant extension to the original Android emulator. The key of extension is to support various sensors, trace replay, and power tracking in the emulation process. The Android emulator [8] does not track power states of hardware components or emulate common sensor devices. Moreover, it does not support feeding pre-collected sensor traces as input for emulation. This precludes the possibility of emulating a sensing app over users' real traces. Other existing emulators such as [9] also do not support power emulation of sensing apps.

We optimize PowerForecaster to make its emulation fast and its trace collection power efficient. First, reliable power estimation requires to use long traces, but this makes emulation very time-consuming. The emulation by default takes as much time as the length of original traces. Emulation in advance is not a viable alternative due to huge combinations of users, sensing apps, and their version updates. PowerForecaster achieves fast emulation by two techniques: (1) fast-forwarding replays in a way to capture changes in power states only and (2) parallelizing replays on a number of emulator instances. Our evaluation shows that emulation with 18-hour long traces is completed within half of a minute, with a small estimation error (6-7 percent). Second, collecting sensor traces on a user's phone incurs nontrivial power overhead. PowerForecaster minimizes necessary data collection while keeping estimation accuracy high by applying a balanced duty-cycling policy based on our empirical experiences. Users do not need to repetitively collect sensor traces. Instead, once the traces are collected, they are reused for various sensing apps.

We further address the problem of dealing with large-scale emulation requests from worldwide deployment, which is an important issue not covered in our previous paper [5]. Considering the number of mobile users and sensing apps, the amount of server resources required to process worldwide requests would be tremendous. To this end, we devise a novel selective emulation approach, namely *similar segment skipping* to significantly reduce the server-side cost for power emulation. The key idea is from our empirical observation that mobile sensing apps take user behavior as an input of the internal logic and their power consumption is mostly determined by how the logic behaves. Upon a user's request, our method sorts out the parts of data traces expected to show the highly similar power consumption for a target app and avoids performing power emulation for those parts. Our evaluation shows that it saves on average 60 percent of the emulator instance usage for three sensing apps.

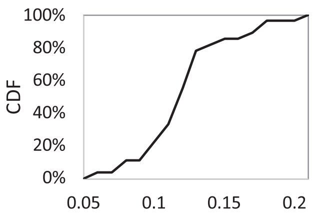
The contributions of this paper are summarized as follows.

- First, we develop an accurate power estimation system based on behavior-aware power emulation for sensing apps, which is the first-of-its-kind system providing user-specific power estimation at pre-installation time.
- Second, we present a set of system optimization for fast power emulation and energy-efficient trace collection to make our system practical.
- Third, we present a novel selective emulation approach to minimize the server-side resource use needed to process a large number of emulation requests.
- Last, we evaluate the performance and overhead of the system through extensive experiments.

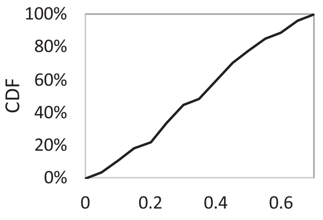
This paper is organized as follows. Section 2 discusses the related work. Section 3 presents the motivating study to show the impact of user behavior on the power consumption of sensing apps. Sections 4 presents the design of PowerForecaster. Section 5 presents data collection and Section 6 details the user behavior-aware power emulation. Section 7 discusses server-side resource optimization. Section 8 presents our experimental results and Section 9 provides discussion on limitations and future works. Finally, Section 10 concludes the paper.

2 Related Work

Mobile Sensing Apps and Energy Optimization. A new class of mobile apps, so-called mobile sensing apps, have emerged



(a) GPS activation ratio



(b) Sound sensing activation ratio

Fig. 2. CDF of activation time ratio.

during the last decade [1]. The mobile computing community have proposed lots of mobile sensing apps [1], [2], [10], [11]. Much effort has been put towards multiple dimensions, such as improving recognition accuracy and optimizing power consumption involved in continuous sensing. Another notable dimension was on proposing common platforms to provide system support for mobile sensing apps [12], [13], [14].

Power Profiling and Modeling. Power profiles and models constitute important baselines for energy optimization. Extensive efforts have been put to build accurate power models of mobile apps and phone H/W [15], [16], [17], [18], [19], [20], [21]. We adopt those models for accurate power estimation. Similar to our emulation-based approach, WattsOn provides a developer tool to emulate apps' power use in development environments [16]. 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. Another important application of mobile apps' power estimation is to diagnose abnormal battery-drain problems due to bugs and misconfigured apps [22], [23], [24], [25], [26]. They help users spot the causes of abnormal battery drain and take counteractions. We complement these works with forecasting the expected power impact of sensing apps, helping users make an informed decision in advance.

Human Battery Interaction. Literature on human-battery interaction on mobile devices reports battery-charging behavior [27], [28], user perception of battery interface [28], [29], user behavior changes upon battery awareness [30], and user-interactive charging [31]. Their focus mainly remains in conventional mobile apps, rather than newly rising issues with continuous sensing apps. As an early attempt, our previous work [4] studied users' concerns about running continuous sensing apps, which exhibit varying battery drain patterns depending on user behavior. We found that continuous sensing apps often embarrasses users as their battery usage largely deviates from what they understand based on their experiences with conventional mobile apps. It proposed a novel tool to provide more realistic battery drain information by taking account of user behavior factors while running sensing apps. Unlike this work, PowerForecaster focuses on providing power impact of sensing apps at pre-installation time and thereby helps users make a better-informed decision.

Mobile App Testing. Several automated testing frameworks for mobile apps have been proposed [32], [33], [34]. They use a monkey tool to generate streams of UI events on target apps running on an emulator and analyze runtime properties such as app crashes and page contents. Our system differs 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.

3 Motivation

Continuous sensing apps may cause inconsistent battery consumption among users, and thereby making users distrust the developer's estimates of power consumption. It is mainly from diversity in individual user behaviors, 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; they use low and high power sensors selectively depending on the user context [10], [35], [36]. How often and long the user's activities trigger the conditions has a significant impact on actual power use. 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 [37]. For instance, a sensing app can obtain 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 homemaker in her 50s). We deployed a datalogging app on their phones. The app collects data for 12 hours a day (from 10 AM to 10 PM), but actual collection time varied due to battery depletion. We analyzed data collected for more than 8 hours a day.

Applications. We consider two sensing apps inspired by previous works: (1) MyPath, a location tracker [38] and (2) ChatMon, a conversation monitor [36]. MyPath records the GPS trace of a user every 10 seconds. GPS sensing is triggered only when the user is moving which is recognized by low power accelerometer sensing. ChatMon uses sound sensing and processing to monitor speakers and conversation turns. It triggers sound sensing only when its user is close to someone else, which is detected by Bluetooth scans every 2 minutes.

Effect of Physical Activities. We first looked into the effect of physical activities. Fig. 2a shows the cumulative distribution of GPS activation time ratio of MyPath. The ratio is computed as the ratio of GPS activation time to total execution time. The average ratio per user ranges from 5.7 to 20.2 percent (mean: 12 percent, SD: 3.2 percent). Assuming that MyPath runs 12 hours a day, the top user activates GPS 1.5 hours more than the bottom. Similarly, we show the ratio of sound sensing activation time of ChatMon, which monitors conversations with the top 5 frequently encountered people. Fig. 2b shows the cumulative distribution of sound sensing activation time ratio. The average ratio varied from 4.8 to 67.1 percent (mean: 34.9 percent, SD: 18.2 percent). The top user would activate 3.5 more hours of sound sensing than the average user, assuming that ChatMon runs 12 hours a day. This would cause a huge difference in the power use of users' phones.

Effect of Phone Uses. We further examined the effect of resource sharing with ChatMon. We analyzed the net increase in CPU activation time caused by wakelocks ChatMon

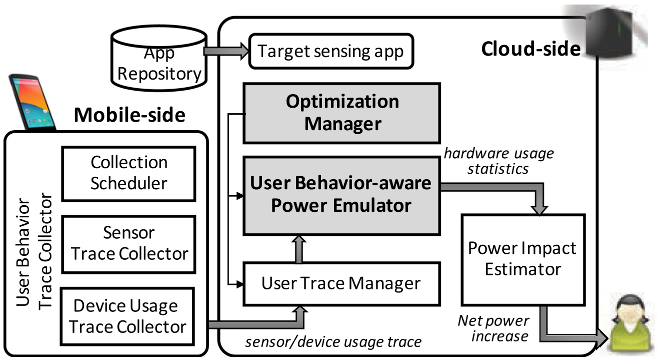


Fig. 3. PowerForecaster architecture.

acquired. 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 percent of the total duration. However, P6's CPU was solely activated by ChatMon for 15 percent of the total duration, while 22 percent was activated for P13. This difference results from that P6 used his phone for other purposes for longer time than P13 did, so P6's CPU was activated more already by other apps. Second, for the other case, P8, P15, and P16 exhibited similar sound sensing activation time, about 40 percent. However, they showed fairly different net increase in CPU activation time: 30, 40, and 33 percent, respectively. This demonstrates that different phone use behaviors would affect net power consumption of a sensing app differently due to the sharing effect.

4 PowerForecaster Design

4.1 Design Goals

Installation-Free. PowerForecaster relieves tiresome trials of repeated installation caused by sensing apps' unknown battery use. To this end, it needs to provide estimated power consumption to users before they install and use the apps.

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

Latency. Estimation should be completed in a short time to serve requests from users on the fly.

Coverage. PowerForecaster aims at providing power estimation without modifying app binaries; any requirement to change the app would remarkably reduce its utility.

Overhead. PowerForecaster should minimize the overhead on users' mobile phones, especially for collecting their sensor traces and device usage traces required for power emulation.

Scalability. PowerForecaster should be scalable to deal with large-scale emulation requests.

4.2 Power Emulation Approach

We propose a *user behavior-aware power emulation* approach. Its 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 factors such as physical activities and phone usage, which significantly affect the power consumption of a target app. (2) It estimates the app's power use without any knowledge on its internal logic or prior power profiling. The emulator tracks hardware use of the app by executing its executable without analyzing its source code. (3) It considers the shared use of hardware components with other already installed and used mobile apps on a user's phone by reproducing their resource use.

There have been other approaches to estimate mobile apps' power use. However, it is difficult to apply them to our target environment where we need accurate power estimation for sensing apps. *Development-time estimation*, direct measurement with a power meter or model-based schemes [15], [16], hardly reflects individual user behavior that considerably affects sensing apps' power use. Another potential approach, *collaborative power estimation*, shares the power impact of sensing apps obtained from users who are already using the apps. The estimated power usage from users with similar behaviors is provided to a new user; a similar concept is proposed in [22] for energy diagnosis. However, this requires a large user pool to find similar users due to a lot of combinations of behavioral factors affecting power use. We believe PowerForecaster can complement this approach, as it works based on personal traces without depending on a large number of users.

4.3 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 8.3 for example). Fig. 3 shows the architecture of PowerForecaster. It is comprised of two major components, a mobile-side trace collector and a cloud-side power emulator. Prior to power estimation requests, the mobile-side collector collects user behavior traces in advance. The traces are uploaded to and managed in the cloud server. Upon a user 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 (See Section 5). The *sensor trace collector* records sensor data that captures users' physical activities. The *device usage trace collector* logs the usage of hardware components by existing apps, which are potentially shareable with a new sensing app. The collected traces are uploaded to a cloud server only when the phone is on Wi-Fi and charging not to interfere with its usual use. They are used for various sensing apps later. The *collection scheduler* manages the schedule of the data collection for energy efficiency (Section 6.5).

Cloud-Side. Upon a power estimation request, the *user behavior-aware power emulator* runs the target app's executable while replaying the sensor and device usage traces to reproduce the execution environment (Sections 6.1 and 6.2). At the same time, it monitors the hardware usage 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 (Section 6.3). The *optimization manager* manages the emulation schedules to achieve the optimization in terms of the emulation delay (Section 6.4) and server resources (Section 7).

TABLE 1
Sensor Data, System Calls, 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(), disable Sensor()	

4.4 System Scope and Limitations

Target Apps. Our target apps are sensing apps that continuously run in the background, such as Google Fit [39] and Accupedo [3]. Specifically, we focus on their autonomous sensing services that repetitively perform sensing tasks controlled by built-in sensing logic 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 have controllability on how long they use the apps and thereby on the apps' power consumption to some extent. For sensing apps, continuous sensing tasks are a major cause of power drain. UI activities consume relatively less power as they are likely to run for a short time compared to the whole operation time of sensing apps.

Tracked Hardware Components. We currently consider hardware components commonly used by sensing apps for power estimation. They include inertial sensors, GPS, Bluetooth and Wi-Fi scans, microphone, and CPU. (See Section 6.3 for details.) We have not yet considered other components such as display and network as those are used less commonly in sensing apps and hardly have an impact on the overall power use. Section 8 shows that our prototype estimates the power use of various sensing apps with an average error of 6.6 percent, considering the aforementioned components. Our system can be further extended to include additional components by leveraging power models proposed in [16], [17].

Selection of User Traces. Accurate power estimation in PowerForecaster requires collecting 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 shown that user behaviors have patterns [40], [41], [42]. 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 work, however, we focus on developing a power emulation system that precisely estimates net power increase of an app given proper user behavior traces. For the current PowerForecaster prototype, we assume that mobile-side user trace collection is performed for a couple of days. We leave capturing users' behavioral patterns from a long period of user traces as future work.

Scalability to Heterogeneous Phones. It is well known that building power models specific to individual phone models is needed to accurately track apps' power use. This raises a scalability issue because a variety of phone models are released in today's markets. As the core technique of our power emulation is based on pre-built power models, the same issue will arise when our system is widely deployed. Note that our current prototype incorporates power models for Nexus S and Nexus 5.

To be optimistic, popular smartphone models take considerable market share worldwide. Top-5 popular iPhone models and top-20 Android ones take 72 percent[1] and 23.3 percent[2] of iPhone and Android market share, respectively. We believe that 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. We can consider other approaches for less common models. For instance, most hardware chipset manufacturers provide power profiles of different hardware states, and those profiles can be utilized for our system. In case such information is not available or the power consumption is not stable across the same phone models [15], [18], we can leverage prior works to automatically self-construct power models of unknown phones [18], [19]; these works achieve an accuracy of 90-95 percent. 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.

5 User Behavior Data Collection

Below we detail the user behavior data collected by the trace collector.

Sensor Trace. The *sensor trace collector* collects three types of information concerning the sensor data that could be relevant to users' physical behaviors. First, it records time-stamped sensor data samples from frequently used target sensors. Second, it records Android system events related to the sensors, e.g., *gps_event_started*, to ensure correct replays of sensing apps with internal triggers from such events. Last, it hooks and logs the system calls and callbacks to/from the Android framework/kernel, in order to reproduce power states in sensor devices as if they were from users' real situations. For example, the time to activate GPS largely depends on whether a user is outdoors or not. Table 1 shows sensor data, events, and system calls that we collect.

Device Usage Trace. For accurate power emulation, it is essential to consider shared use of resources [5], [37]. Emulating every app concurrently running with a target sensing app would accurately trace the shared use of various hardware components. However, it will seriously burden the cloud-side emulators. Foreground apps add another

1. https://www.statista.com/statistics/606147/iphone-model-device-market-share-worldwide/
2. https://www.appbrain.com/stats/top-android-phones

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

challenge of logging and replaying interactive user inputs. For light-weight yet accurate estimation of sharing effects, we *record only the device usages of existing apps*, so that the shared use of hardware among concurrent apps is efficiently accounted without directly executing the apps.

To this end, the *device usage trace collector* records the apps' requests to various hardware components, e.g., wake-locks for CPU, with timestamps. The emulator replays those requests while emulating the sensing app to identify the hardware components being shared by the target sensing app and other apps. Android exposes a narrow set of APIs for apps to request access to hardware components. The collector logs such API calls and their parameters. Table 2 shows what our prototype collects in more detail.

6 User Behavior-Aware Power Emulation

Existing mobile emulators [8], [9] are limited for the power emulation of sensing apps. First, they are primarily designed for the testing and debugging of functional operations of mobile apps and lack testing non-functional performance requirements of apps such as power usage. Second, they cannot reproduce sensing and subsequent processing affected by users' behavior, which is a key aspect of evaluating the power use of sensing apps.

To address the problems, we design and develop a *user behavior-aware power emulator* as shown in Fig. 4. We built it by extending the Android emulator significantly. It operates through the following three steps.

Pre-Emulation Stage. A new power emulation request goes to the *emulator manager*. It first initiates emulator instances while obtaining sensor and device usage traces from the *trace manager* along with the executable of the target app; note that multiple instances are created to accelerate emulation (See Section 6.4 for details). Each instance then installs the app, receives a part of user traces, and adjusts its system clock to synchronize the time with the traces. In this step, the *emulator manager* highly optimizes resource use of the emulator by skipping emulation of user traces with minimal impact on the power usage of the app (See Section 7).

Power-Emulation Stage. The emulator instance executes the target app. Upon the app's requests on sensor data, the *sensor emulator* mimics the operation of sensors based on the pre-collected sensor traces (Section 6.1). Also, the *device usage replayer* reproduces other apps' device use based on the device usage trace (Section 6.2) to emulate concurrent execution of the target app with other apps. During the execution, the *hardware usage monitor* tracks system calls to use hardware components, e.g., sensors and CPU, and generates hardware usage statistics.

Post-Emulation Stage. The *power impact estimator* computes the net increase in power consumption by the target app using the hardware usage statistics (Section 6.3). We adopt a system call-based approach to estimate the power consumption – i.e., using power profiles obtained by offline profiling [17]. We support the following hardware components used by sensing apps: *CPU*, *GPS*, *accelerometer*, *gyroscope*, *magnetometer*, *microphone*, and *Bluetooth*/*Wi-Fi scans*.

6.1 Sensor Emulation

The sensor emulator is unique in that it reflects user's physical behaviors, which significantly affect the power consumption of sensing apps. It feeds the user's real sensor traces to the target app at an accurate timing and rate as the app requests. It also emulates the hardware states of the corresponding sensor devices to accurately account for power estimation.

Prior sensor simulation tools [43], [44] do not serve our purpose. SensorSimulator [43] provides a custom library that relays pre-collected sensor data from a host PC to a mobile emulator. However, it requires modification of the app. ReRan [44] 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, the app has to be executed during sensor data collection, which cannot be used at pre-installation time. More importantly, both of the simulators do not emulate the power state of sensor devices.

Our sensor emulator is located between the Android framework and the kernel, which facilitates to mimic real

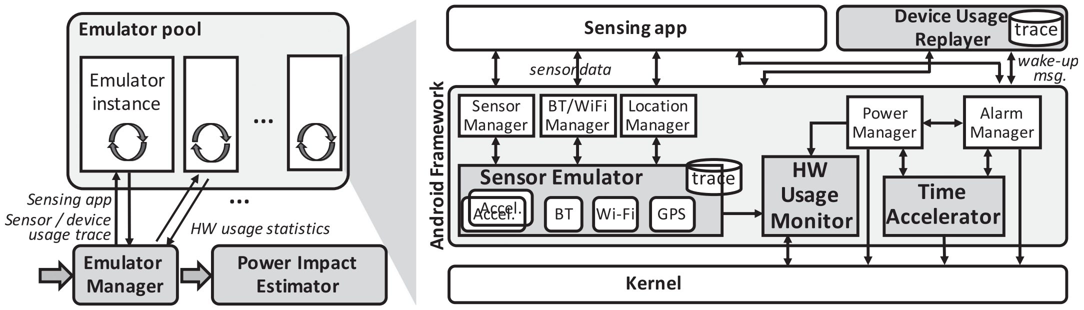


Fig. 4. Architecture of user behavior-aware power emulator.

sensor devices' operation and states. It provides apps with pre-collected sensor data and replays sensor-related system events for app operations. Also, it tracks the changes in expected power states based on the sensor-related system call and callback logs to and from the kernel.

Accelerometer, Gyroscope, etc. The sensor emulator hooks 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 *Wifi-Manager*. Upon the request, it retrieves scan logs with a timestamp closest to the request time from the collected trace and forwards them to the app. To ensure the app's 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.

6.2 Device Usage Replay

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

Since API calls to access hardware components often operate as a pair, e.g., *acquire()*/*release()*, both calls should be included in the trace for accurate power estimation. However, one of the pair could be omitted at times because of duty-cycled data collection (Section 6.5) or the segmentation of device usage traces for parallel execution (Section 6.4). 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 sets its timestamp to the beginning of the trace.

6.3 Estimation of Power Impact

The *hardware usage monitor* collects the hardware usage statistics during emulation, i.e., a collection of system calls and their timestamps. Sensor-related system calls are captured in the sensor emulator (See Table 1). We also 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 records CPU utilization of the target sensing app from */proc/stat* until a release request. We scale the CPU utilization to compensate for the different CPU performances between the server and the smartphone as in [16].

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 that app uses and $P^D_{with\_app}$ and $P^D_{without\_app}$ are the power consumption of D with and without app, respectively.

The power consumed by the hardware components is estimated with a system call-based power estimation [17]. We built a finite state machine (FSM)-based power model for each component based on associated system calls. 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 [17]. To make the power model, we profile the power states of the components for each system call using a power meter. $P^D_{with\_app}$ is computed based on the 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.

6.4 Acceleration of Emulation

A key challenge of emulation-based power estimation is 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 100 Hz for 5 sec.

To address the challenge, we leverage unique characteristics of continuous sensing apps and develop three acceleration mechanisms: *parallel execution*, *idle time skipping*, and *progressive estimation*. Continuous sensing apps operate by repeating cycles. While the operations could be stateful within a cycle, they might be stateless in between. This implies the potential to parallelize the emulation process. Also, sensing apps wake 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 users movement with 20-ms accelerometer data and sleeps again if no movement. We can safely skip such long idle time to accelerate the emulation. More details on the acceleration techniques can be found in [5].

6.5 Energy-Efficient Trace Collection

The trace collector collects sensor and device usage traces for future power estimation requests. A challenge is to collect the sensor traces, desirably at high sampling rates, at a low power cost. We applied a widely used duty-cycling technique to reduce the power cost of the trace collector. The question is how far we can increase the cycle with a minimal decrease in accuracy. Fig. 5 shows the estimation errors with

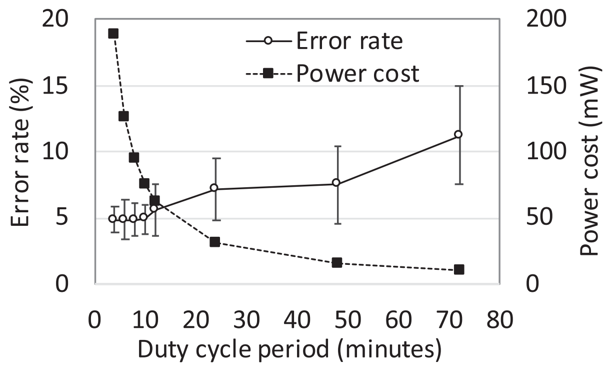


Fig. 5. Effect of duty-cycling; bars represent standard deviation.

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 8.3 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 5). 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 percent even with the deviation. Such duty cycling is possible since user's mobility, location and encounter tend to have temporal locality [40], [41], [42].

7 Server-Side Resource Use Optimization

Server-side cost for handling a power forecasting request from a single user would not be significant. Our analysis shows that completing a request for a given user and a sensing app takes about 30 sec on average with a total of 45 emulator instances running parallel on two physical servers when the acceleration techniques are applied [5].

However, if we consider the potential scale for worldwide deployment, the amount of required server resource would be tremendous. Based on the number of app downloads from the Play Store in 2015, we estimate that the worldwide workload requires 7,800 physical servers under the assumption that users would compare three similar apps before installing one, i.e., a total of four power forecasting requests per installation [5]. We believe much more servers will be needed to provide PowerForecaster as a commercial service. Mobile app markets are still growing (the number of Android app downloads increased from 50 billion in 2015 to 90 billion in 2016[3]) and diverse sensing apps keep proliferating including health monitoring, personal assistant and so on. Also, it is preferred to provide the information about more number of similar apps at once for better user experience. Considering these premises, an efficient way to process large-scale requests is essential.

7.1 Basic Idea

The basic idea for saving server-side cost is to reduce the number of data segments that we need for power emulation, which is directly mapped to the number of required emulator instances. As discussed, PowerForecaster divides a trace into multiple segments and parallelizes their power emulation for responsive services. The possibility of the reduction is based on our observation that the segments from the same user behavior exhibit quite similar power consumption. It is reasonable considering that mobile sensing apps take user behavior as an input of the internal logic, e.g., movement monitoring for a pedometer, and the power consumption is mainly determined by how the logic of the app works. If we know which segments would yield similar power consumption before performing the power emulation process, we can skip those segments in emulation, thereby reducing the server-side resource use significantly.

3. http://www.businessofapps.com/data/app-statistics/

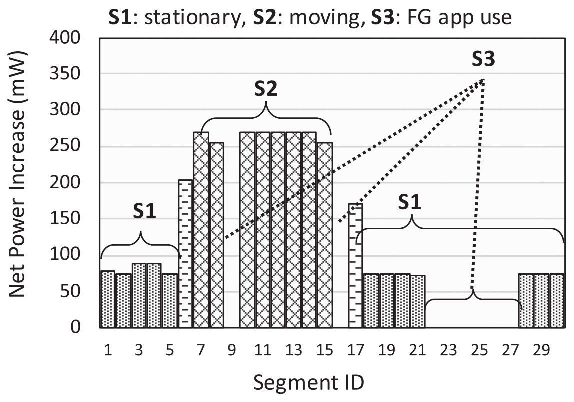


Fig. 6. Net power increase depending on the behavior of a user.

Fig. 6 shows the estimated net increase of power consumption for 30 two-minute segments labelled by corresponding user behaviors. This example shows the case of Scenario-3 running an Accupedo app (in Section 8). As shown in the figure, the segments associated with the same behaviors of a user show similar net power increase. This supports the intuition that the number of emulator instances can be reduced without compromising the accuracy, as long as only a part of segments relevant to distinct user behaviors is emulated. We expect that the effect of the reduction will be non-trivial with the traces collected in real-life situations. While people do many different activities every day, their behaviors tend to remain the same for a certain duration, e.g., taking a lecture for an hour, roaming around in a shopping mall, having lunch in a cafeteria, etc.

A key question is how to group segments associated with the same user behaviors given the user behavior traces. If we know the contexts affecting the power consumption of an app and the sensing pipeline used in the app, grouping segments without the app execution might be possible by analyzing sensor data and the pipeline. In our environment, however, we use app binaries as system inputs, and thus their internal logic is not known. Besides, such an analysis would not be straightforward.

7.2 Similar Segment Skipping

It is challenging to figure out related segments in terms of their power consumption without the actual execution of a target app and avoid unnecessary emulation. First, the power impact of the segments is different depending on the app logic. For example, some changes in acceleration values (or their magnitude) can result in a significant difference of power cost for an app that triggers GPS by user's movement. On the contrary, the same change may hardly increase the power use if the app continuously activates accelerometer, but has no subsequent processing. Second, the device usage trace should

TABLE 3
Example of Element Data Type

Data Category	Data Type	Value Format
Sensor data	Accelerometer	A time series of (x, y, z)
	GPS activation	A list of (start time, end time)
	Bluetooth scan	A list of (timestamp, found device ids)
Device usage	Wakelock status	A list of (acquire time, release time)
	GPS activation	A list of (start time, end time)

be considered simultaneously together with the sensor data trace to take the power-sharing effect into account and derive net power increase. Third, the user behavior traces including sensor data and foreground app usage are composed of heterogeneous types of data. Table 3 shows an example of element data types of the trace. For example, accelerometer data are numerical data, i.e., a series of timestamped acceleration values along the three axes, whereas wakelock data consists of pairs of activation start time and end time. Also, the sampling rates of the data are different from each other.

To address the problem, we develop a *similar segment skipping method*. Its key idea is to build an app-specific similarity function using pre-stored user behavior traces when a new app is registered to the system (Section 7.2.1). Then, upon a power forecasting request, the method clusters segments based on the similarity defined by the function, performs power emulation with a subset of segments for each cluster, and estimates the net power increase from the emulation results (Section 7.2.2).

7.2.1 Building Power Similarity Function Prior to Request

We develop a novel similarity function that takes the power consumption of segments into consideration. This implies that higher similarity scores are given to segments which yield more similar power consumption. When a similarity function for a specific app is once made, it can be generally used for all users' requests.

To deal with heterogeneous types of data in user behavior traces, we use a different similarity measure for each type of data, combine them into one measure, and perform clustering with the combined measure. This is a common approach for clustering heterogeneous data sets [45]. More specifically, we define the similarity of an app between Seg_i and Seg_j as follows:

$$Sim(Seg_i, Seg_j) = \sum_{k=1}^{N} C_k Sim_k(Seg_i.T_k, Seg_j.T_k) + C_{N+1},$$

where Sim_k is the similarity for data type T_k, $Seg_i.T_k$ is the type T_k data of segment i, C_k is the coefficient for type T_k, and C_{N+1} is a constant. Below we present how to compute the coefficients C_k and a constant C_{N+1} as well as the similarity for each data type, Sim_k.

Calculating Coefficient and Constant Values. It is important to carefully determine coefficient values based on the power impact of each data type on the given app. A simple way is to compute the overall similarity value by the sum or multiplication of the similarity values per data type regardless of their power impact. From our observation, however, there are some cases that the power consumption is different even though the overall similarity of two different segments is high. This is mainly because the different types of data affect the power consumption of a sensing app differently. For example, consider a case of an app using accelerometer and GPS. Given two data segments, the power consumption may largely differ from each other if the similarity of GPS usage is low even though the similarity of accelerometer data is very high.

To reflect the power impact of each type of data, we define the above-mentioned similarity function, $Sim(Seg_i, Seg_j)$, based on the similarity between the power values of two segments, i.e., $1 - \frac{|P_i - P_j|}{max(P_i, P_j)}$, where P_i is the power consumption of Seg_i. This shows that the similarity will be close to 1 if two segments yield similar power consumption and 0 otherwise. Then, we apply a regression approach to determine the coefficient values.

The operation for deriving the coefficient values is as follows. First, we segment given user behavior traces. In our implementation, we used one 18 hours-long trace, i.e., 540 two-minute segments, for the input of the regression; the system can use more traces for better accuracy. Second, we estimate the power consumption of those segments using the power emulator. Third, we enumerate all possible pairs of the segments, i.e., ${}_{540}C_2$ and calculate the power similarity values of those pairs, $Sim(Seg_i, Seg_j)$. Fourth, we compute the similarity values, Sim_k of each type of data for all pairs of segments. Last, we calculate the coefficient and constant values by applying the linear regression.

Similarity for Each Data Type (Sim_k). For the time-series sensor data such as accelerometer and gyroscope, we use multi-dimensional cosine similarity. For the activation data like Wakeock and GPS activation, we first convert it to an activation list and then apply a cosine similarity function. The activation list is a list of bins; each bin represents a time range with the same duration and contains the activation ratio within a time range of the bin. In our current implementation, we set the bin size to one second. For example, if the WakeLock is activated from 0.7 to 2.5 sec, the activation list will be [0.3, 1, 0.5].

7.2.2 Efficient Power Estimation Upon a Request

Segment Clustering. Upon a user's request, PowerForecaster divides his/her own behavior trace into two-minute segments and performs clustering of the segments. For the clustering algorithm, we use hierarchical clustering because there is no apriori information about the number of clusters required. We define the distance of two segments as the reciprocal of their similarity, i.e., $\frac{1}{Sim(Seg_i, Seg_j)}$.

Overall Power Impact Estimation. After clustering, we randomly choose a single segment per cluster and perform the power emulation only for the selected segments. The overall power impact of a trace is estimated by the sum of the estimated power cost of the selected segments weighted by the number of segments in their cluster.

8 Evaluation

We implemented the PowerForecaster prototype, which consists of two main components. For the mobile side

TABLE 4
Summary for Five User Behavior Scenarios

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

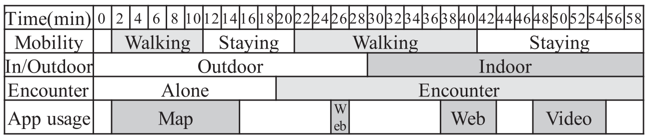


Fig. 7. User behavior scenario 4.

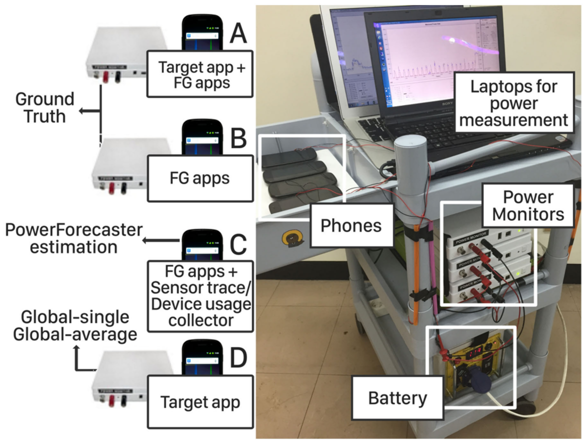


Fig. 8. Experimental setup.

component, we developed the sensor trace collector as an Android service. We also modified the Android system to track device usage and existing apps' hardware usage. For the server side, we built the power emulator based on the Android emulator. We also implemented the user trace manager and power impact estimator in Java.

Overview. We conduct extensive experiments with the PowerForecaster prototype in the following aspects. First, we evaluate the accuracy of the net power estimation of PowerForecaster with Monsoon power monitor in diverse user scenarios (Section 8.2). Second, we investigate the system overhead of PowerForecaster on the mobile and cloud side (Section 8.2.3). Third, we perform the real deployment study and observe the performance of PowerForecaster under two-week long user behaviors (Section 8.3). Last, we study how the similar segment skipping mechanism reduces the server cost of power emulation (Section 8.4).

8.1 Evaluation Setup

Phones and Servers. We used Nexus S (Android 4.1.2) and Nexus 5 (Android 4.4.4) phones for our experiments; we used Nexus S phones by default. For emulation, we used 12 desktop servers (with i7-2600k CPU and 16 GB RAM), each of which was configured to run one or more emulator instances in parallel. Handling a single power estimation request costs 5-10 percent of the average CPU utilization per each emulator instance. We discuss the server-side resource use in more details in Section 8.4. We did not apply optimization techniques by default.

Sensing Apps. We mainly used three sensing apps, i.e., a commercial pedometer app, *Accupedo* [3], and the two research apps we developed, *MyPath* and *ChatMon* [5]. The hardware components that Accupedo uses are an accelerometer and CPU. While diverse sensing apps have been proposed in the research community, only a few types are commercialized, e.g., pedometers. To cover more diverse hardware usage, we selected MyPath and ChatMon. The former uses an accelerometer, GPS, and CPU and the latter uses Bluetooth, a microphone, and CPU.

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

Global-single measures the power consumption 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 consumption 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 consumption while running simultaneously 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. We performed scenario-based experiments to evaluate power estimation accuracy. We crafted multiple one-hour scenarios with four parameters: mobility, encounter, indoor/outdoor status, and phone usage. We determined the parameter values and their combinations based on real user data described in Section 3. Table 4 summarizes the five scenarios used for the experiments. Fig. 7 describes *Scenario 4* with detailed sequences of user activities and phone usages. Other ones are made similarly.

Measurement and Trace Collection Setup. To make comparison fair, we should assure that the same user behaviors are applied while evaluating all the alternative techniques including the ground truth. We devise a setup to easily conduct experiments over a set of alternatives. Fig. 8 shows the setup with four phones (Phone A, B, C, and D) and three power monitors. We used Phone A and B for ground truth measurements; Phone A and B measures power consumption while running a target sensing app with existing apps and while running existing apps only, respectively. We calculated the ground truth by subtracting the power consumption of Phone A from that of B. For the estimation by

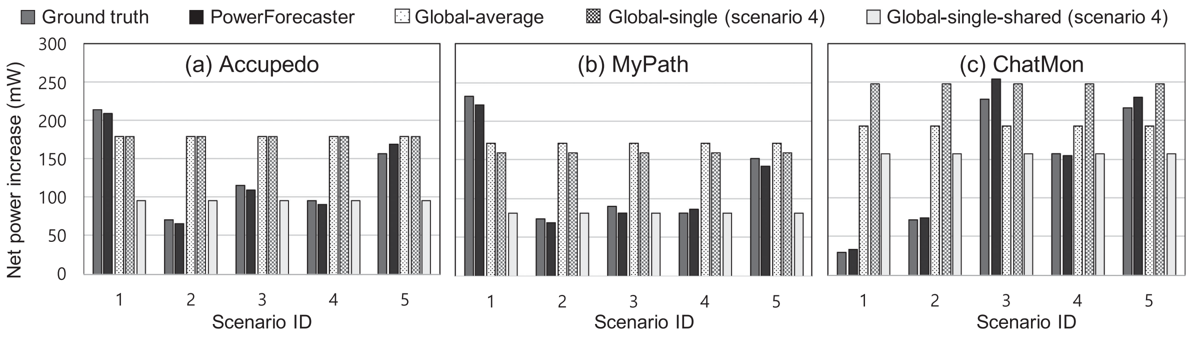


Fig. 9. Accuracy of net power estimation by PowerForecaster.

PowerForecaster, 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. For measurement, an experimenter drove the cart with the phones and power monitors following each scenario. We used a script to run the same apps (a web browser, a video player, an email client, and a map) at the same timings.

8.2 Performance Analysis of Power Emulation

8.2.1 Accuracy of Net Power Estimation

We measure the power estimation of PowerForecaster for three sensing apps in five scenarios. The experimental results show that net power increases by sensing apps are accurately estimated. Fig. 9 depicts that the average error rate is 6.6 percent. More specifically, the average error for Accupedo, MyPath, and ChatMon is 5.3, 6.8, and 7.7 percent, respectively. This indicates that PowerForecaster captures the power use of the target sensing apps accurately for various scenarios. Interestingly, the net power increase is different across the scenarios even for the same app because user activities and phone usages are different. For example, the ground truth of the net power increases by the ChatMon app is within a range between from 29 to 227 mW.

Different from PowerForecaster, the accuracy of the three alternatives is much lower. For example, Global-average shows the average error rate of 99 percent across all scenarios and apps. The average error of Global-single(Scenario4) is 116 percent, i.e., misestimating the app's power cost by more than double the ground truth. In the case of Global-single with Scenario2 and Scenario3, the average error rates are reported as 59 and 128 percent, respectively. The results show that Global-single estimates the net power increase of a target app based on a single specific scenario only and the approach itself is error-prone for different scenarios. Surprisingly, the error rates of Global-single(Scenario4) are high even for the same scenario (Scenario4), e.g., 86 percent for Accupedo. The main reason is that Global-single does not take the effect of resource sharing with the existing app usage into consideration. Global-single-shared(Scenario4) takes the sharing effect into account, but still produces the high average error rate, i.e., 60 percent. For instance, Global-single-shared(Scenario2) and Global-single-shared(Scenario3) reports the error rate of 45 and 83 percent, respectively. When considering the sharing effect, the error becomes 0 percent if the same scenario is chosen for the baseline and estimation, but the error is made largely and differently depending on the behavior modeled in other scenarios.

We further investigate the applicability of PowerForecaster to two more commercial pedometer apps, NoomWalk and Pedometer2.0, and a more device, Nexus 5. Similarly, PowerForecaster provides an accurate net power estimation. First, the estimations for NoomWalk and Pedometer2.0 with Scenario 4 are 11 and 105 mW while the ground truths are 9 and 103 mW, respectively. Second, it achieves 6 percent of average error rate using Nexus 5 with the three sensing apps: Accupedo, MyPath, and ChatMon with Scenarios 4, 1, and 2, respectively.

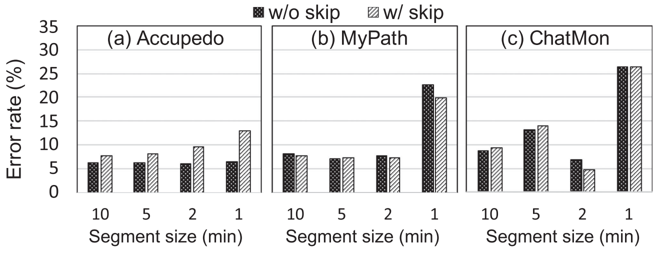


Fig. 10. Effect of parallel execution and idle time skipping.

8.2.2 Effect of Emulation Acceleration

We investigate the effect of the emulation acceleration regarding the tradeoff between time and accuracy. We chose 10, 5, 2, and 1 min for the segment sizes of the parallel execution.

Parallel Execution with Idle Time Skipping. The experimental results show that PowerForecaster still achieves a low error rate even with the acceleration. Fig. 10 shows the results with different segment sizes; in the figure, w/o skip and w/ skip represent the case when the parallel execution is applied without and with idle time skipping, respectively. More specifically, w/o skip and w/ skip produces the average error rate of 10.4 and 11.2 percent, respectively. The error increases slightly compared to the case without the acceleration but is still reasonable. The estimation accuracy is affected by the segment size. For example, the error rates of Accupedo are less than 10 percent regardless of the segment size, but MyPath and ChatMon show the relatively large errors with the 1-min segment. This is mainly due to the characteristics of their sensing logic.

We further examine the elapsed time, i.e., the period to be taken to finish the emulation. We measure the elapsed time for three sensing apps and all five scenarios with 2-min segments. Interestingly, the emulation is mostly completed within 20 sec. More specifically, the emulation of 30 percent of segments for Accuepedo is completed within 10 sec and

TABLE 5
Mobile-Side Cost

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 (296)	0.8	0.5
sensor (full)	34.6	32 (390)	26 (315)	1.1	0.6

the time for 70 percent of segments of MyPath is 20 sec. ChatMon shows a similar trend to Accupedo.

Progressive Estimation. We study the error of the progressive estimation over time when 2-min segment parallel execution is applied with idle time skipping. The error of MyPath and Accupedo quickly goes down and stabilizes within 10 sec. After 20 sec, the error of MyPath becomes 9 percent while the error of Accupedo does 19 percent. The error almost saturates in 30 sec. For ChatMon, it takes relatively longer time, 60 sec, for the saturation, but differently depending on the scenarios. The error of ChatMon with Scenario3 and Scenario4 is saturated within 12 sec with the error of 5 percent. 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.

8.2.3 System Overhead for Power Emulation

Mobile-Side Cost. We investigate the energy overhead and storage size on the mobile side. We omitted the network cost since PowerForecaster uploads the trace only while the smartphone is connected to Wi-Fi and charged. For the measurement, we consider two sensor sets, one with GPS, accelerometer, and Bluetooth (a basic set), and the other with all available sensors on the smartphone we use for experiments (a full set). Table 5 shows the cost when 1/12 duty cycle is applied, which we obtain from our experiment in Section 6.5; we set the sampling rate of accelerometer, light, proximity, magnetometer to delay_fastest, the sampling rate of gyroscope to delay_game, and the sensing interval of GPS, BT, 3G, WiFi to 1 min. We calculate the decrease of the battery life under the assumption that the battery life with the fully charged battery is 15 hours. The battery capacity of Nexus S and Nexus 5 is 1,500 and 2,300 mAh, respectively. The results show that the overall power costs for both sensor sets are not significant. The battery life is expected to decrease by one hour on Nexus S even with a full sensor set. The power cost to collect the sensor data during the collection is 390 mW, but the average power becomes 32 mW due to duty cycling. The mobile-side cost to collect the device usage trace is negligible. It is important to note that the overall mobile-side cost is acceptable because, if PowerForecaster collects the trace once, it can use the collected trace for various different apps.

Server-Side Cost. The major operations and costs on the server-side as follows. First, prior to a request, the user behavior traces are stored and managed in the emulator manager. The cost of the network and storage is not large because the trace for a single day of a user is about 35 MB at the maximum. Second, once a request is given, PowerForecaster emulates power behaviors with the pre-segmented traces on multiple phone emulators in a parallel way. A single emulator instance on our server consumes about 5 percent of CPU and 400 MB of memory. We figure out that the cost is mostly to emulate a virtual phone image and the additional cost to replay the trace is marginal. Last, the power impact is estimated based on the hardware usage statistics collected from the emulators. The calculation takes less than one sec.

8.3 Real Deployment Experiment

Experimental Setup. 6 graduate students and 1 researcher (P1) were recruited on campus in Nov. 2014. For a two-week period, each participant replaced his/her primary phone with a Nexus S phone that we provided. They installed their own SIM cards and all the apps they use. To collect their daily sensor data without affecting their phones'usual power consumption, we provided each of them with another Nexus S that just keeps collecting sensor data. To ensure that both phones are under the same user behaviors and environmental conditions, we taped them up for each participant to carry them together. Every day, we collected the full sensor data from 8 am until 2 am the next day. Each participant was compensated KRW 200,000 (USD 179). In the first week, we collected prior information to estimate power impact, e.g., sensor traces, device usage traces, hardware usage statistics, and battery levels. Upon beginning the second week, each participant installed one of three sensing apps, Accupedo, MyPath, or ChatMon. Since then we measured the decrease of the battery life due to this extra sensing app they began using. For ChatMon, two students from the same project group were recruited and their ChatMons were configured to detect each other.

User-Friendly Power Impact Estimation. We further process $netP_{app}$ in a more user-friendly way, by converting it into the expected decrease in the battery life (in hours) of the user's phone by the formula:

$$\begin{aligned}\text{decrease(app)} &= \text{battery\_life}_{without\_app} - \text{battery\_life}_{with\_app} \\ &= \frac{capacity}{P_{without\_app}} - \frac{capacity}{(P_{without\_app} + netP_{app})},\end{aligned}$$

where $capacity$ is the phone's full battery capacity and $P_{without\_app}$ is the average power use of the phone without the target sensing app. $netP_{app}$ is the net power increase of the app outputted by our emulator. For 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 [22].

Results. Table 6 shows the battery-life decrease per sensing app, per participant. The decrease was 12.1 hours on average. Even a commercial app, Accupedo reduces battery life by 5.3-14.7 hours.

To study the estimation accuracy, we asked the participants to select two days during the first week, each with different IDs in Table 7. Note that it is impossible to measure exact estimation accuracy without completing 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 7 shows PowerForecaster's observed battery life and estimated future life, taking the user trace on that day into the input. It also shows

TABLE 6
Summary of Real-Deployment Experiment

ID	Age	2nd-week sensing app	1st-week avg. battery life (h)	2nd-week avg. battery line (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 7
Estimation of Battery Life

ID	TraceID	Battery life (h)	Estimated (future) battery life (h)	2nd-week avg. battery line (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**	

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 percent for 10 cases out of 14. Even for Accupedo, the accuracies were over 90 percent for all six cases.

Few outlying cases exhibit relatively lower accuracy, e.g., for P5 with the trace 5-4. P5 mentioned he did not use the phone as usual nor move much since it was a weekend day. Our data indicates that P5's battery life was about 20 hours on average during the first week, but 33 hours in trace 5-4. This observation advises us to separately estimate power behaviors on weekends. We will extend PowerForecaster to make progressive classification of user's daily life patterns.

Now we examine the estimation accuracy with all the optimization techniques applied. For 1/12 duty cycling, we sampled 2-min data out of every 24-min traces. Power emulation was performed on the sampled traces with 2-min segment parallel execution and idle time skipping. Fig. 11 shows the progressive estimation errors over time with trace 4-1 and 7-3, compared to the baseline using the whole trace and no acceleration technique. Even sampling only 1/12 of the trace enables PowerForecaster to achieve quite low errors of 8.5 and 3.4 percent, respectively. This supports our intuition that the sampled data well-represents the rest thanks to the context continuity. The trace 4-1 exhibits decreasing errors over time, 13 and 10 percent at 20 and 30 sec, respectively. For trace 7-3, it converges to 3 percent after 5 sec.

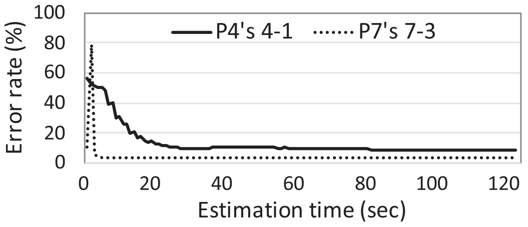


Fig. 11. End-to-end performance with P4's 4-1 and P7's 7-3.

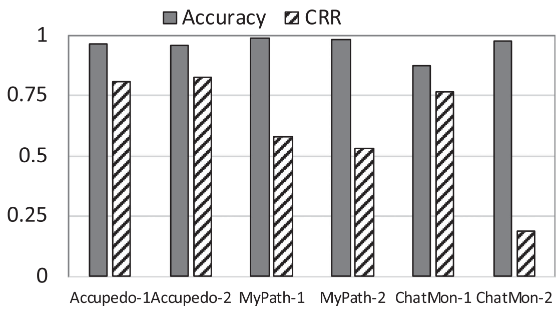


Fig. 12. Performance of similar segment skipping.

8.4 Effect of Similar Segment Skipping

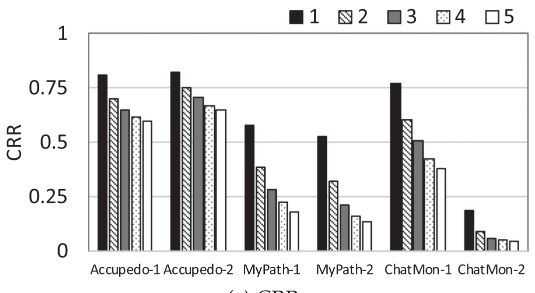
We further evaluate how effectively PowerForecaster reduces the cost of power emulation with a similar segment skipping mechanism. We measure its effectiveness regarding power estimation accuracy and server load reduction. The power estimation accuracy represents the accuracy of power estimation with the similar segment skipping compared to the power estimated with a full data trace as ground truth. For the server load reduction, we use the cost reduction ratio (CRR), which is one minus the ratio of the number of emulator instances used for power estimation to the number of originally required instances. For the current implementation, the number of instances needed to process an 18-hour-long trace is 540 as we take a 2-minute segment size for parallel execution. For example, if our mechanism uses 135 instances only for the power emulation, CRR is 0.75.

For the evaluation, we used the dataset collected in our real deployment experiment; we used two user behavior traces for each sensing app and conducted the cross-validation, i.e., using one trace to make a similarity function and the other to apply the similar segment skipping method for evaluation, and vice versa. We currently use a single 18-hour-long trace for the similarity function, but PowerForecaster can leverage more traces if available. It is important to note that, if the similarity function is once made, it can be generally used for different user requests.

For the power estimation after segment clustering, PowerForecaster randomly selects one segment in each cluster and estimates the overall power impact using the power values derived from the emulation of the selected segments. We repeated such a clustering and estimation process for 100 times under the same trace and parameter settings, and reported the average results.

8.4.1 Overall Performance

Fig. 12 shows the accuracy and CRR for each sensing app; each app has two cases of cross-validation. The


(a) CRR

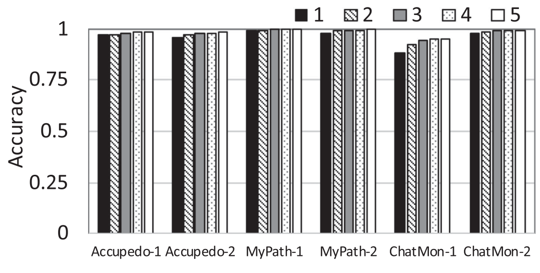
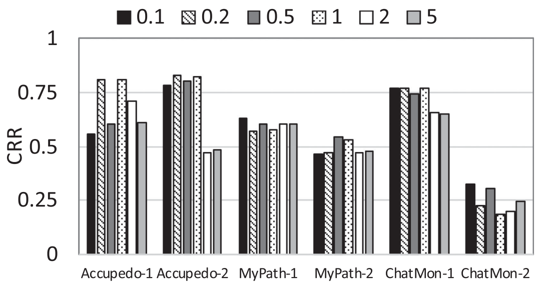

(b) Accuracy

Fig. 13. Effect of number of segments per cluster.


(a) CRR

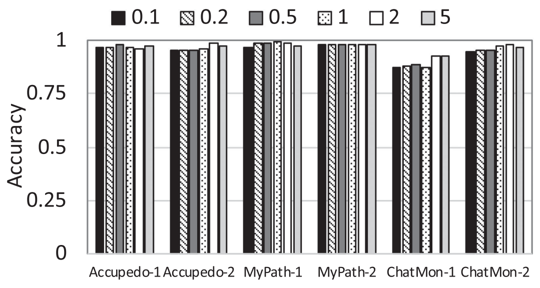

(b) Accuracy

Fig. 14. Effect of bin size (seconds).

experimental results show that our similar segment skipping method significantly reduces the server load, while preserving the high accuracy of the power estimation. Compared to the estimation with the full-length trace, our method used only 40 percent of the originally required instances on average, i.e., 60 percent of CRR, but still achieved 96 percent of the power estimation accuracy.

The CRR varies depending on the sensing app, while the accuracy remains mostly high. The average CRR of the Accupedo is 0.81, but that of MyPath and ChatMon is 0.55 and 0.48, respectively. It is because the segments are clustered differently depending on the hardware components that the app uses. Even for the same sensing app, CRR is different depending on the composition of the contexts in the user behavior trace used for making a similarity function. For example, the CRR of ChatMon-1 and ChatMon-2 is 0.77 and 0.19, respectively. The trace used for making a similarity function of ChatMon-2 contained a limited set of user behaviors and thus the generated similarity function does not well estimate the distance between segments for a similar, but unseen pattern. We again emphasize that our method guarantees the high accuracy of the power estimation regardless of the clustering performance. Also, the CRR can improve if our service is deployed and a sufficient amount of traces is available for making a similarity function.

8.4.2 *Performance Breakdown*

Effect of the Number of Segments per Cluster. We investigate the effect of the number of segments per cluster, used for the power emulation; we used one segment for each cluster by default. Figs. 13a and 13b show CRR and the accuracy while increasing the number of segments per cluster from 1 to 5, respectively. The results show that more number of segments does not much contribute to accuracy improvement. This implies that our similarity function between segments well reflects their power impact and thus the segments are well clustered in terms of the power estimation. However, CRR decreases considerably. For example, the CRR of Accupedo-1 decreases from 0.81 to 0.60, while the accuracy increases by 0.01, i.e., 0.97 to 0.98. Note that the CCR does not decrease linearly proportional to the number of segments because some clusters have less number of segments than the specified number. From this result, we can see that a small number of segments to emulate is sufficient for accurate power estimation, while guaranteeing high CRR.

Impact of Bin Size. We investigate the effect of the bin size on the power estimation. The bin size is time length to be used to generate an activation list and set to one second for the current implementation. Based on this, we can expect that the smaller bin size increases the number of bins for a segment and accordingly is able to define the activation list in a more fine-grained way. This implies that the smaller size may help more accurate power clustering, but less contributes to the server cost reduction. Figs. 14a and 14b show CRR and the accuracy, respectively. The results show that there is no common trend in CRR with varying bin size. This is mainly because the user behavior has a temporal locality and thus the activation list was made similarly regardless of the bin size.

9 Discussion

Daily Variation of User Behaviors. Our focus in this paper is on providing an accurate estimate of power impact due to a target sensing app given users' behavioral traces of a specific day. However, one may argue that the real power impact in the future cannot be well represented by the estimation for a past specific day. From our study, we observe both possibilities and limitations. The deployment study result discussed in Section 8.3 shows that the estimation accuracy is reasonably good. This implies that users are likely to have similar behavioral patterns over days. At the same time, there are also cases that show noticeable deviation from the estimate for some users. From the results, it might be difficult to provide a single representative estimate based on one-day-long traces in the face of daily variations of behaviors. A potential approach to address the problem is to collect user traces for several days (possibly periodically) and make an estimate in a more detailed form, e.g., a reasonable range rather than a single value. While this can increase the cost imposed on the mobile side, it might be a useful option for users who are not bothered by collecting traces for a longer period of time. Moreover, if it is possible, it may be able to detect their behavioral patterns and model their behaviors as studied in [40], [41], [42]. Once such a model is derived, we can provide power impact of sensing apps more systematically.

Power Estimation of Advanced Techniques for Sensing Apps. To estimate power consumption of sensing apps, we focused on the local processing model, i.e., sensing apps relying on their built-in logic to sample sensor values and process the data repeatedly. However, some apps could rely on more heterogeneous processing models, such as offloading to the cloud [46] or utilizing separate low-power processors [47]. To address such apps, it is required to track their network usage and/or take account of different power models for network usage and low power processors. We will extend our system to track network-relevant factors and reproduce them in the emulator [16] as well as incorporate additional power models for low power processors.

Privacy. Given the nature of PowerForecaster analyzing users' sensor and device usage data on the cloud, a next question to ask would be about privacy concerns. Ultimately, such concerns would be about benefit-risk tradeoff, i.e., does the system offer sufficient benefit given the risk levels [48]. Many users already let cloud services manage their private data, e.g., photos and emails for convenience and usability reasons. In fundamental, however, privacy is still a very subjective and sensitive issue [49]. To relieve the concern, a lot of research efforts have been put, e.g., allowing users to control the access and granularity of the data [50], securely encrypting the data [51], and leveraging the secure authentication [52]. Our system will adopt such solutions.

10 Conclusion

We present PowerForecaster, a system that provides personalized power impact of mobile sensing apps prior to installation and actual use. We show that individual user behaviors are important to understand power impact of sensing apps. We build a user behavior-aware power emulator that accurately estimates net power increase by sensing apps based on user's behavioral traces. Furthermore, we develop a novel solution to saving the resource use required for emulation in order to efficiently deal with large-scale requests for power impact prediction. We implement and extensively evaluate the PowerForecaster prototype in terms of estimation accuracy, speed, and the reduction of resource use.

Acknowledgments

The authors thank Sangwon Choi, Pillsoon Park, Seungpyo Choi, and Seungchul Lee for their valuable aid in implementation, data collection, and analysis. This work was in part supported by the National Research Foundation of Korea (NRF) grant (No. 2017R1C1B1010619, No. 2019R1C1C1006088), Next-Generation Information Computing Development Program (NRF-2017M3C4A7066473) through the National Research Foundation of Korea (NRF) funded by the Korea government (MSIT). An earlier version of this paper was presented at SenSys 2015 [5].

References

[1] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell, "A survey of mobile phone sensing," *IEEE Commun. Mag.*, vol. 48, no. 9, pp. 140–150, Sep. 2010.
[2] Y. Lee, S. Iyengar, C. Min, Y. Ju, S. Kang, T. Park, J. Lee, Y. Rhee, and J. Song, "MobiCon: A mobile context-monitoring platform," *Commun. ACM*, vol. 55, no. 3, pp. 54–65, 2012.
[3] Accupedo. [Online]. Available: https://play.google.com/store/apps/details?id=com.corusen.accupedo.te, Accessed on: Oct. 26, 2018.
[4] C. Min, C. Yoo, I. Hwang, S. Kang, Y. Lee, S. Lee, P. Park, C. Lee, S. Choi, and J. Song, "Sandra helps you learn: The more you walk, the more battery your phone drains," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, 2015, pp. 421–432.
[5] C. Min, Y. Lee, C. Yoo, S. Kang, S. Choi, P. Park, I. Hwang, Y. Ju, S. Choi, and J. Song, "PowerForecaster: Predicting smartphone power impact of continuous sensing applications at pre-installation time," in *Proc. 13th ACM Conf. Embedded Netw. Sensor Syst.*, 2015, pp. 31–44.
[6] C. Min, Y. Lee, C. Yoo, S. Kang, I. Hwang, and J. Song, "PowerForecaster: Predicting power impact of mobile sensing applications at pre-installation time," *GetMobile: Mobile Comput. Commun.*, vol. 20, pp. 30–33, 2016.
[7] C. Min, S. Lee, C. Lee, Y. Lee, S. Kang, S. Choi, W. Kim, and J. Song, "PADA: Power-aware development assistant for mobile sensing applications," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, 2016, pp. 946–957.
[8] Android emulator. [Online]. Available: https://developer.android.com/studio/run/emulator, Accessed on: Oct. 26, 2018.
[9] iOS simulator. [Online]. Available: https://help.apple.com/simulator/mac/current/#/deve44b57b2a, Accessed on: Oct. 26, 2018.
[10] Y. Lee, C. Min, C. Hwang, J. Lee, I. Hwang, Y. Ju, C. Yoo, M. Moon, U. Lee, and J. Song, "SocioPhone: Everyday face-to-face interaction monitoring platform using multi-phone sensor fusion," in *Proc. 11th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2013, pp. 375–388.
[11] Z. Feng, Y. Zhu, Q. Zhang, L. M. Ni, and A. V. Vasilakos, "TRAC: Truthful auction for location-aware collaborative sensing in mobile crowdsourcing," in *Proc. IEEE INFOCOM*, 2014, pp. 1231–1239.
[12] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song, "SymPhoney: A coordinated sensing flow execution engine for concurrent mobile sensing applications," in *Proc. 10th ACM Conf. Embedded Netw. Sensor Syst.*, 2012, pp. 211–224.
[13] S. Kang, J. Lee, H. Jang, Y. Lee, S. Park, and J. Song, "A scalable and energy-efficient context monitoring framework for mobile personal sensor networks," *IEEE Trans. Mobile Comput.*, vol. 9, no. 5, pp. 686–702, May 2010.
[14] Y. Lee, C. Min, Y. Ju, S. Kang, Y. Rhee, and J. Song, "An active resource orchestration framework for PAN-scale, sensor-rich environments," *IEEE Trans. Mobile Comput.*, vol. 13, no. 3, pp. 596–610, Mar. 2014.
[15] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, 2010, pp. 105–114.
[16] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," in *Proc. 18th Annu. Int. Conf. Mobile Comput. Netw.*, 2012, pp. 317–328.
[17] 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. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 29–42.
[18] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proc. 9th Int. Conf. Mobile Syst. Appl. Serv.*, 2011, pp. 335–348.
[19] F. Xu, Y. Liu, Q. Li, and Y. Zhang, "V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics," in *Proc. 10th USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 43–55.
[20] L. Guo, T. Xu, M. Xu, X. Liu, and F. X. Lin, "Power sandbox: Power awareness redefined," in *Proc. 13th EuroSys Conf.*, 2018, Art. no. 37.

[21] L. Sun, H. Deng, R. K. Sheshadri, W. Zheng, and D. Koutsonikolas, "Experimental evaluation of WiFi active power/energy consumption models for smartphones," *IEEE Trans. Mobile Comput.*, vol. 16, no. 1, pp. 115–129, Jan. 2017.

[22] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma, "Carat: Collaborative energy diagnosis for mobile devices," in *Proc. 11th ACM Conf. Embedded Netw. Sensor Syst.*, 2013, Art. no. 10.

[23] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, "Edoctor: Automatically diagnosing abnormal battery drain issues on smartphones," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 57–70.

[24] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proc. 10th Int. Conf. Mobile Syst. Appl. Serv.*, 2012, pp. 267–280.

[25] H. Wu, S. Yang, and A. Rountev, "Static detection of energy defect patterns in android applications," in *Proc. 25th Int. Conf. Compiler Construction*, 2016, pp. 185–195.

[26] M. Wan, Y. Jin, D. Li, J. Gui, S. Mahajan, and W. G. Halfond, "Detecting display energy hotspots in android apps," *Softw. Testing Verification Rel.*, vol. 27, no. 6, 2017, Art. no. e1635.

[27] D. Ferreira, A. K. Dey, and V. Kostakos, "Understanding human-smartphone concerns: A study of battery life," in *Proc. Int. Conf. Pervasive Comput.*, 2011, pp. 19–33.

[28] A. Rahmati and L. Zhong, "Human–battery interaction on mobile phones," *Pervasive Mobile Comput.*, vol. 5, no. 5, pp. 465–477, 2009.

[29] D. Ferreira, E. Ferreira, J. Goncalves, V. Kostakos, and A. K. Dey, "Revisiting human-battery interaction with an interactive battery interface," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, 2013, pp. 563–572.

[30] K. Athukorala, E. Lagerspetz, M. Von K ügelgen, A. Jylhä, A. J. Oliner, S. Tarkoma, and G. Jacucci, "How carat affects user behavior: Implications for mobile battery awareness applications," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2014, pp. 1029–1038.

[31] L. He, Y.-C. Tung, and K. G. Shin, "iCharge: User-interactive charging of mobile devices," in *Proc. 15th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2017, pp. 413–426.

[32] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2014, pp. 204–217.

[33] K. Lee, J. Flinn, T. J. Giuli, B. Noble, and C. Peplin, "AMC: Verifying user interface properties for vehicular applications," in *Proc. 11th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2013, pp. 1–12.

[34] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2014, pp. 190–203.

[35] D. H. Kim, Y. Kim, D. Estrin, and M. B. Srivastava, "SensLoc: Sensing everyday places and paths using less energy," in *Proc. 8th ACM Conf. Embedded Netw. Sensor Syst.*, 2010, pp. 43–56.

[36] C. Luo and M. C. Chan, "SocialWeaver: Collaborative inference of human conversation networks using smartphones," in *Proc. 11th ACM Conf. Embedded Netw. Sensor Syst.*, 2013, Art. no. 20.

[37] N. D. Lane, Y. Chon, L. Zhou, Y. Zhang, F. Li, D. Kim, G. Ding, F. Zhao, and H. Cha, "Piggyback CrowdSensing (PCS): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities," in *Proc. 11th ACM Conf. Embedded Netw. Sensor Syst.*, 2013, Art. no. 7.

[38] J. Paek, J. Kim, and R. Govindan, "Energy-efficient rate-adaptive GPS-based positioning for smartphones," in *Proc. 8th Int. Conf. Mobile Syst. Appl. Serv.*, 2010, pp. 299–314.

[39] Google fit. [Online]. Available: https://play.google.com/store/apps/details?id=com.google.android.apps.fitness, Accessed on: Oct. 26, 2018.

[40] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proc. 8th Int. Conf. Mobile Syst. Appl. Serv.*, 2010, pp. 179–194.

[41] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi, "Understanding individual human mobility patterns," *Nature*, vol. 453, no. 7196, pp. 779–782, 2008.

[42] A. G. Miklas, K. K. Gollu, K. K. Chan, S. Saroiu, K. P. Gummadi, and E. De Lara, "Exploiting social interactions in mobile systems," in *Proc. Int. Conf. Ubiquitous Comput.*, 2007, pp. 409–428.

[43] Sensor simulator. [Online]. Available: https://developer.samsung.com/technical-doc/view.do?v=T000000132, Accessed on: Oct. 26, 2018.

[44] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and touch-sensitive record and replay for android," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 72–81.

[45] A. Abdullin and O. Nasraoui, "Clustering heterogeneous data sets," in *Proc. 8th Latin Amer. Web Congr.*, 2012, pp. 1–8.

[46] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow, "SociableSense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing," in *Proc. 17th Annu. Int. Conf. Mobile Comput. Netw.*, 2011, pp. 73–84.

[47] M.-R. Ra, B. Priyantha, A. Kansal, and J. Liu, "Improving energy efficiency of personal sensing applications with heterogeneous multi-processors," in *Proc. ACM Conf. Ubiquitous Comput.*, 2012, pp. 1–10.

[48] J.-S. Lee and B. Hoh, "Sell your experiences: A market mechanism based incentive for participatory sensing," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun.*, 2010, pp. 60–68.

[49] W. Itani, A. Kayssi, and A. Chehab, "Privacy as a service: Privacy-aware data storage and processing in cloud computing architectures," in *Proc. 8th IEEE Int. Conf. Depend. Autonomic Secure Comput.*, 2009, pp. 711–716.

[50] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[51] G. Wang, Q. Liu, and J. Wu, "Hierarchical attribute-based encryption for fine-grained access control in cloud storage services," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 735–737.

[52] H. Dinesha and V. K. Agrawal, "Multi-level authentication technique for accessing cloud services," in *Proc. Int. Conf. Comput. Commun. Appl.*, 2012, pp. 1–4.

Chulhong Min received the PhD degree in computer science from KAIST, in 2016, and joined Nokia Bell Labs, in 2017. He is a research scientist with Nokia Bell Labs, Cambridge, United Kingdom. He is an associated editor of the *ACM Proceedings on Interactive, Mobile, Wearable and Ubiquitous Technologies*. His research interests include mobile systems and services, machine learning, the Internet of Things, and human behavior modeling.

Youngki Lee is an assistant professor of the Computer Science and Engineering Department, Seoul National University. His research focuses on mobile and sensor systems to enable always available and highly enriched awareness of human behavior and contexts. Also, he has been building innovative life-immersive mobile applications in various domains such as daily healthcare, childcare, and education. He served as the program and general chair of ACM UbiComp 2018 and as a member technical program committees and organizing committees of various prestigious conferences.

Chungkuk Yoo received the PhD degree from KAIST, in 2018. He is a research staff member with IBM. Within the broad spectrum of mobile computing, his research interests lie in mobile applications for in-situ social interaction in real world. He is also interested in mobile systems for visual sensing and recognition.

Inseok Hwang received the PhD degree in computer science from KAIST, in 2013. He is a research staff member with IBM and an IBM master inventor. His research interests lie in AI-powered interactive systems embodied in physical and edge space, driven by novel orchestration of embedded AI, mobile/IoT/robotic platforms, and human-computer interaction. He has been serving on the program committees of various premier conferences. He is a recipient of the Best Paper Award in ACM CSCW 2014.

Younghyun Ju received the PhD degree in computer science from KAIST. He is currently a researcher with Hyundai Motor Company. His research interests include mobile and pervasive computing, technologies for mobility services, and large-scale distributed systems.

Junehwa Song received the PhD degree in computer science from the University of Maryland at College Park. He is a professor with the School of Computing, KAIST. His research interests include mobile, IoT, and ubiquitous systems, Internet technologies, and multimedia systems.

Seungwoo Kang received the PhD degree in computer science from KAIST, in 2010, and joined the Korea University of Technology and Education (KOREATECH), in 2015. He is an assistant professor with the School of Computer Science and Engineering, KOREATECH. He is an associated editor of the *ACM Proceedings on Interactive, Mobile, Wearable and Ubiquitous Technologies*. His research interests include mobile and ubiquitous computing, the Internet of Things, and mobile systems. He is a member of the IEEE.