Fixing Sensor-Related Energy Bugs through Automated Sensing Policy Instrumentation

Yuanchun Li, Yao Guo, Junjun Kong, Xiangqun Chen
Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
{ liyc14, yaoguo, kongjj07, cherry }@sei.pku.edu.cn

Abstract

As mobile applications (*apps*) become more and more complex, many apps contain various energy bugs, which may cause energy wastes that might reduce the battery life to as short as several hours. Among them, sensor-related bugs such as *sensor data underutilization* is one of the most common energy bugs. Instead of trying to detect these energy bugs, this paper proposes a method to fix sensor data underutilization automatically through instrumentation of existing apps. App-specific energy-aware sensing policies can be written to the apps via an automated instrumentation process, which can also be customized by users if needed. The proposed technique is easy to apply as it does not need to modify the operating system or the apps. At the same time, it also works for existing legacy apps, which makes it practical and feasible for a wide-range of mobile apps. Experimental results on popular Android apps show that we are able to achieve significant energy savings through automated instrumentation and rebuilding the targeted apps.

Keywords

Mobile applications, sensors, instrumentation, energy optimization, Android

1. Introduction

Smartphones have become more and more popular since the introduction of iPhone and Android-based devices. Compared to traditional phones, smartphones have more powerful functions and are capable of performing complex computations, while equipped with various sensors such as cameras and GPS. More and more complex mobile applications (or simply *apps*) are running on smartphones, which could reduce the battery life to as short as several hours.

Energy inefficiencies (or energy bugs [1]) in mobile apps is one of the main reasons why smartphones have a very short battery life. For example, researchers have identified that many apps contain various energy bugs such as no-sleep bugs [2], energy leaks [3], and sensor-related energy black holes [4][5]. Among all kinds of energy bugs, sensor-related energy bugs is one of the most common problems, while the most common case is *sensor data underutilization*. For example, a navigator app may keep fetching GPS coordinates even when GPS signal is weak or imprecise; a pedometer app may keep sampling accelerometer at a high frequency even when the smartphone is motionless. Sensor data underutilization is a common problem because developers are either unaware of the power difference at different rate levels, or simply ignore them to reduce development efforts.

Detecting these energy bugs is only a first step towards solving the battery life problem. In order to fix these energy bugs, we usually need to report these bugs to app developers who owns the source code, and wait for them to take action and release a new version. However, this might not always be possible if your favorite app does not have an active maintainer. Even somebody is willing to modify the code, it could also become a lengthy process. This paper explores these issues in a different direction, searching for an approach to automatically fixing these energy bugs.

In this paper, we attempt to solve the sensor data underutilization issues from a different angle: *fixing sensor data underutilization issues automatically without the help of developers*. Our proposed solution does not depend on the availability of source code, instead we exploit the reverse engineering capability of Android apps and fix the energy bugs through instrumentation. App-specific energy-aware sensing policies will be instrumented into the apps through automatic patching, repackaging and re-installation. In a typical usage scenario, a user only needs to click a button to transform his apps into an energy-aware version. A more technology savvy user could also define his/her own policies to save more energy.

Our approach optimizes sensing energy of real-world apps by attaching app-specific sensing policies. First, we define different contexts for different apps. For example, navigator apps use INDOOR and URBAN contexts, while pedometer apps use MOTIONLESS and MOVING contexts. Then we identify different actions taken by each app in the same context. For example, some sensor-related games need to keep frequently sampling motion sensor data because the apps need to rapidly react to sudden motion in the MOTIONLESS context. We also provide an interface for developers to define app-specific contexts and context-aware sensing policies, and provide an interface for users to choose appropriate policies to automatically optimize the sensing apps.

Compared to existing work, our proposed solution has several advantages. First, it has the ability to automatically transform legacy sensing-related apps into energy-efficient versions without needing source code. Second, different frequency scaling policies can be easily applied to mobile apps, either statically or dynamically. Finally, it can be applied to a wide range of mobile sensors on modern smartphones.

We have implemented an instrumentation framework on Android to demonstrate the applicability of the proposed technique. We are able to add adaptive sensing abilities into existing apps such as navigators and pedometers. The evaluation results with volunteer users show that sensing sampling frequencies of a pedometer app can be reduced by more than 90%, while energy consumption can be reduced by more than 50% on average.

2. Background and Related Work

2.1. Sensing in Mobile Apps

Modern smartphones are equipped with multiple sensors such as accelerometer, gyroscope, magnetic field sensor, digital compass, proximity sensor, GPS, microphone, etc.

Many mobile apps are using multiple sensors. Our analysis of the top 500 apps in each of the 27 app categories in Google Play shows that over 55% of apps use some kinds of sensors through various Android sensing APIs (shown in Figure 1).

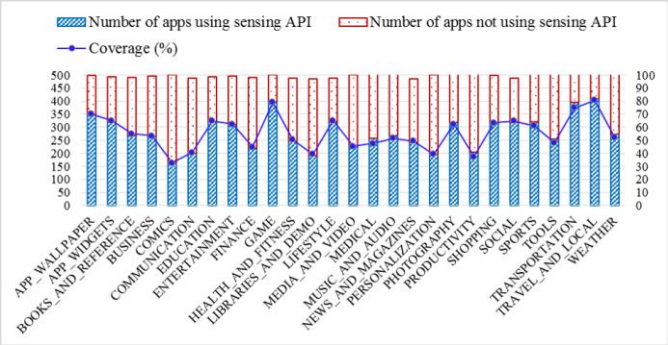

Figure 1: Distribution of mobile apps using sensing APIs in different categories.

In Android, in order to fetch data from a sensor, an app needs to assign a sampling rate level (FASTEST, GAME, UI, NORMAL[1]) while registering a sensor event listener to the targeted sensor. After registration, sensors work at the assigned rate level and pass sensor events (which carry sensor raw data) to the callback function of the sensor event listener.

Different sampling rate may affect both sensor access latency and sensing power. For example, at the FASTEST level, sensors usually have the lowest latency and highest power, while at the NORMAL level, sensors usually have the highest latency but lowest power. Our measurements show that the access latency and power consumption can both be varied by up to an order of magnitude[2], which shows that there exists great potential for energy optimization with proper tradeoffs between latency and power.

2.2. Energy Bugs

Energy bugs were first introduced by Pathak *et al.* [1]. They have also characterized no-sleep bugs [2] and use data flow analysis to detect no-sleep sensor energy bugs.

ADEL [3] uses dynamic taint analysis to detect and isolate a different class of energy bugs called *energy leaks*. An energy leak is the use of energy on activities that never directly or indirectly influence user-observable outputs on a smartphone.

GreenDroid [4][5] makes further efforts to detect sensor related energy black holes. They extend the definition of sensor energy bugs and study *sensor data underutilization* as a kind of sensor energy inefficiency bugs. GreenDroid can explore the state space of an app by systematically executing the app using Java Path Finder.

Instead of detecting sensor data underutilization problem, our work attempts to fix them automatically.

2.3. Sensing Energy Optimization

There have been many related work on improving energy efficiency of sensing-related and context-aware apps. One way to do this is to reduce energy consumption of high-power sensors in order to achieve whole system energy saving, such as in LEAP [6]. Hardware-based techniques like co-processor off-loading are also used for sensing energy saving, such as LittleRock [7] and DSP.Ear [8]. At a higher level, context-aware energy optimization can substitute high-energy sensors with low-cost sensors to improve sensing energy cost [9][10][11][12].

Sampling frequency scaling is efficient for sensing energy reduction because power consumption rises significantly as the rate level rises. Apps are often able to save power without too much precision loss by scaling down the sampling frequency level. Several recent studies [13][14][15][16] proposed adaptive sensing techniques as a trade-off between energy and accuracy. One typically method is to scale the sampling rate based on contexts. For example, Paek *et al.* [14] and Lin *et al.* [13] lower location sampling frequency when GPS is inaccurate in some urban area. LiKamWa *et al.* [17] apply an aggressive standby mode in which they choose an optimal clock frequency to reduce energy consumption.

Some other work have tried to solve the sensing energy problem from the developers' perspective. For example, Kansal *et al.* [18] presents a new programming abstraction allowing developers to specify the latency, accuracy and battery constraints.

These existing approaches mainly achieve energy saving with sensor-level and context-based optimization techniques. Sensor-level optimization mostly requires modification to the firmware or even to the hardware. Context-based energy optimization approaches are mainly based on a common observation: sensor data of a high-power sensor is often underutilized in some context. For example, navigator apps might underutilize GPS sensor data when smartphone is indoor or in urban areas, where GPS signal is weak or imprecise [14]; Pedometer apps might keep wasting motion sensing energy when the smartphone is motionless; Web apps might repetitively send connection requests when the phone is in bad network state. Existing studies focus on the definition and detection of these contexts, and implement sensing optimization policies mostly at the system level (which is hard to customize) or at the app level (which needs modification to the source code).

The existing approaches either deal with specific sensors, require modifications to the system or require new development efforts. In comparison, our approach attempts to fix energy bugs by modifying the mobile apps even when source code is unavailable.

3. Our Approach

3.1. Approach Overview

Our goal is to optimize the energy consumption of apps that rely heavily on sensors, particularly *continuous sensing apps*, which means that in order to maintain its functions, the app must access relevant sensors periodically at a preset sampling rate level (e.g., there are four different sampling rate levels in Android, as specified above). Our basic idea is to help mobile apps to choose the most appropriate sample rate.

[1] Their sampling frequencies are listed in descending order.

[2] We omitted the experiment data here due to space limitation.

Considering the following scenario when driving a car using your smartphone as a navigation device. Smartphones will retrieve GPS data at a fixed frequency, regardless of your driving speed. However, when you are waiting at a traffic light, you do not need to access the GPS data at the same frequency as when you are driving at 120km per hour. The sampling rate could be adjusted dynamically with modification to either the device driver, the sensor access libraries or OS services.

In this paper, we do not want to modify the system, as it might involve complicated updating at the user side. We want to transform a mobile app into an energy-aware version with adaptive sensing as described above.

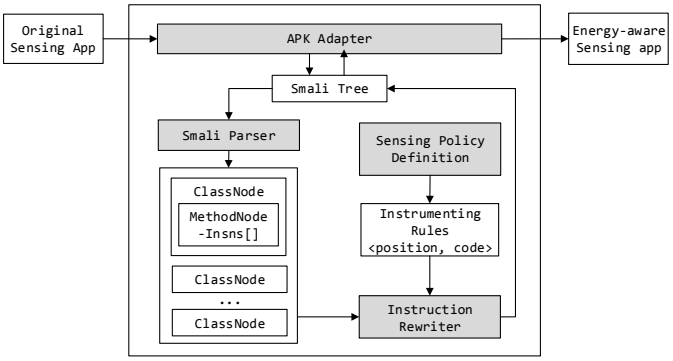

Figure 2: The workflow of the proposed approach.

How do we write these adaptive policies to an existing (Android) app? Figure 2 illustrates the workflow of the proposed approach. The key technique behind our approach is the *app rewriting framework*, which relies on instrumentation to apply energy-aware sensing policies through the energy-aware sensing APIs we defined.

Energy-aware sensing policies can be either defined manually or generated automatically according to the requirements of a specific app. The energy-aware sensing APIs extend the existing Android sensing APIs and enable mobile apps to dynamically change sensor sampling frequency according to the specified sensing policies at runtime. Finally, the app rewriting framework completes the actual rewriting process and transform an existing sensing app into an energy-aware version.

3.2. Energy-Aware Sensing APIs

Android apps follow a typical process when accessing a sensor, which mainly involves two sensing APIs:

SensorManager;->registerListener(...);

SensorManager;->unregisterListener(...);

In a typical sensing app, it first defines a sensor event listener and registers it to the sensor by calling *registerListener*. Once the sensor completes a sampling event, the sensed raw data is passed as an argument of the callback method *onSensorChanged* in the listener. The frequency of sensor data updating depends on a parameter of the *registerListener* method, which can be specified as one of the following: FASTEST, GAME, UI, or NORMAL.

The sensing APIs provided by Android can be easily misused by average developers. The register API tells the Android system to sample sensor data at a certain rate level, and the rate level will never change until the unregister API is called. If we register a sensor event listener with the FASTEST level, the sampling frequency will keep as FASTEST no matter how the sensor data is used. An experienced developer could determine whether the sensor data is efficiently used and scale down the rate level once underutilization is found. However most developers ignore these issues and register the listener only once for convenience.

We define *energy-aware sensing APIs* as adaptive versions of the original sensing APIs in a class named *AdaptiveSensorManager*. The adaptive sensing APIs take exactly the same parameters as the original ones, but they can dynamically change sensor status according to the sensing policy. For example, if we call the adaptive *registerListener* API with the FASTEST rate level, it does not mean the sensor will remain working at the FASTEST frequency all the time. Instead, the sensing frequency will decrease when we found the high frequency is not necessary.

3.3. Energy-aware Sensing Policies

To fix sensor underutilization bugs, we define context-based adaptive sensing policies, which includes two parts:

1. *Context definitions.* Sensor data is underutilized in some contexts and efficiently utilized in other contexts. Contexts should be defined with clear entering criteria and exiting criteria.
2. *Scaling actions.* Scaling actions describe how we scale the sampling frequency in different contexts. For example, sampling frequency should be scaled down when sensor underutilization is detected, while it should be scaled to the original level if the sensor data are used efficiently.

Context determination is the most important part in adaptive frequency scaling. Many existing context-aware approaches make use of sensor data, location, and WIFI signals to determine various contexts. In this paper, we simply use the accelerometer and WIFI signal information to determine the contexts.

For example, we can define a *Sampling Frequency Scaling* policy for pedometer apps. The original pedometer app samples the accelerometer at the FASTEST level even in the MOTIONLESS context, which is a case of sensor data underutilization, thus we can decrease its sampling frequency in the MOTIONLESS context. Similarly, when the smartphone is moving, the variation of accelerometer will exceed a certain threshold, and the context will be switched to MOVING. In the MOVING context, the sampling frequency will be scaled up.

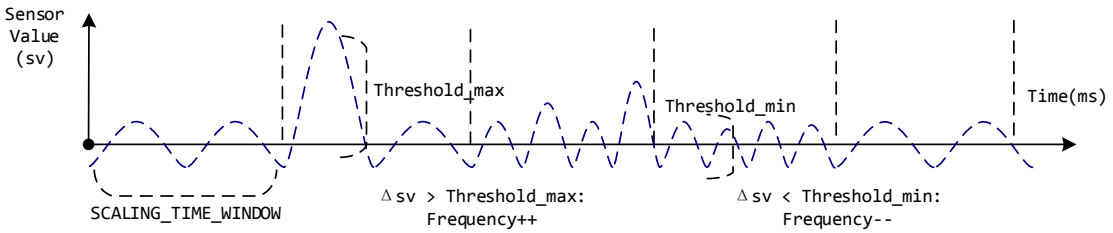

Figure 3: An example of the sampling frequency scaling policy of the *pedometer* app.

Figure 3 presents an example of adaptive scaling for the pedometer app. The app tries to scale frequency every *SCALE_TIME_WINDOW* seconds. At first, the app worked at a low frequency. At the end of the second cycle, the MOVING context was detected and the pedometer started to scale up it sampling frequency. At the end of the fourth cycle, the app enters the MOTIONLESS context and the frequency was decreased.

For another example, we can define a *Block Indoor GPS* policy for navigator apps. For a navigator app, we can simply determine whether the app enters the INDOOR context by checking whether the WIFI AP that the smartphone is currently connected to is in a pre-defined indoor WIFI AP list. When the smartphone is in the INDOOR context, we can turn off GPS positioning to save energy, and vice versa.

A principle of defining contexts is that the context determination cost should be much smaller than the original sensing cost. For example, we make use of the existing accelerometer data to determine the context for the pedometer, and we use cheaper WIFI information for the GPS navigator.

Each app has its specific sensing policy. A sensing policy may be applicable for a set of apps, but there is no universal policy. For example, the Sample Frequency Scaling sensing policy we used for the pedometer app is not applicable for some games using the accelerometer, such as a maze app. Games need to react to users' sudden movement, so it cannot tolerate the long latency caused by scaling down the sampling frequency.

3.4. The App Rewriting Framework

App instrumentation depends on the targeted system. In our case, the targeted system is Android. It is easy to rewrite an Android app. For example, we can dissemble the *classes.dex* file into to the human-readable *smali* format using *baksmali*. Similarly, the *smali* code, after modification, can be converted back to a new app.

As shown in Figure 2, the app rewriting framework for Android apps includes the following steps:

1. Disassemble the *.dex* file to *smali* code.
2. Run static analysis on the *smali* code and identify instrumenting entrances.
3. Insert adaptive sensing library code to appropriate positions in the *smali* code.
4. Assemble the rewritten *smali*, repackage and sign the instrumented application.

Our adaptive sensing library works with a service named *ContextService*. *ContextService* runs in background detecting the smartphone's context and controls the sensing status of sensing apps according to the corresponding sensing policies. It requires the adaptive sensing libraries we added into apps able to communicate with *ContextService*. We solved this by adding a sensing monitor thread to the sensing library, and the thread starts running once the sensor is getting to work. To enable IPC between the sensing monitor threads and *ContextService*, we make use of the AIDL mechanism in Android. We insert an AIDL module to each sensing app and install the *ContextService* using AIDL interfaces, thus sensing apps are able to get current context and sensing policies generated by *ContextService*

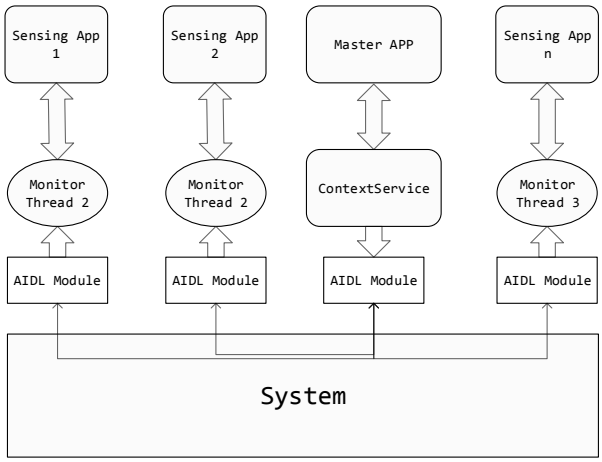

Figure 4: The structure of rewritten sensing apps.

Furthermore, we introduce a master app to let users configure sensing policies for different sensing apps. Smartphone users can select from several recommended pre-defined sensing policies, or define their own policies.

After rewriting, each sensing app has an extra adaptive sensing library compared to the original version. If the master app and *ContextService* are installed as required, the structure of the sensing apps interacting with each other will be the same as shown in Figure 4.

4. Evaluation

In order to demonstrate the performance of sensing policy, we apply the *Sample Frequency Scaling* sensing policy to a pedometer app, and perform a series of experiments with both in-lab tests and real user traces.

In our experiments, we use Google Nexus 5 with Android 4.4 to install and run a pedometer app from Google Play[3]. In lab experiments, we use the Monsoon power monitor to calculate energy consumption and generate a power model of

[3] https://play.google.com/store/apps/details?id=name.bagi.levente.pedometer

the accelerometer at different rate levels. In real user trace study, we calculate how much time the pedometer spent at each rate level, and calculate the total energy consumption according to the power model.

4.1. Lab Experiments

In order to understand the energy savings and whether we are able to maintain the detection accuracy in the original app, we first conducted experiments in a controlled lab environment.

4.1.1. Power Consumption

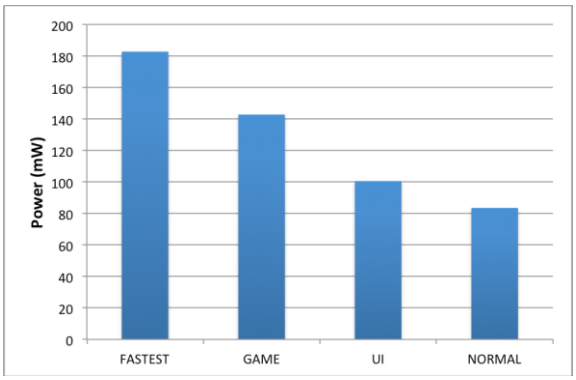

Figure 5: Power consumption of the smartphone, while running pedometer at different rate levels (screen off).

We first measure the power consumption of the whole smartphone with the power monitor. With the help of bytecode rewriting, we control the pedometer to run at different rate levels (with screen off). We measure the power consumption at each rate level.

Figure 5 shows the results. We can see that the smartphone power can be reduced from 180mW to 80mW when we set the accelerometer into the NORMAL (slowest) mode, instead of the FASTEST mode, which shows that it is meaningful to control the sampling rate of accelerators.

4.1.2. Accuracy

In order to measure the accuracy of the pedometer app, we compare the step counts calculated by the modified app, and compare them with the original app.

We ask our volunteers to carry the smartphone with the pedometer running, while at same time counting his own steps. After walking for 200 steps, we record the step counts of the pedometer at 50th, 100th, and 200th step, respectively. We record three groups of step counting data, the first two groups are from the original pedometer, and the last group uses our repackaged pedometer with adaptive sensing enabled. The results are shown in Table 1.

	Original App 1	Original App 2	Optimized App 3
At 50th step	61	57	59
At 100th step	104	117	107
At 200th step	214	221	219

Table 1: Accuracy comparison of the original and energy-optimized pedometer.

We notice that the pedometer tends to over-estimate the number of steps, but the range of errors remains stable after we apply the energy-aware policies in the instrumented app.

4.2. User Trace Study

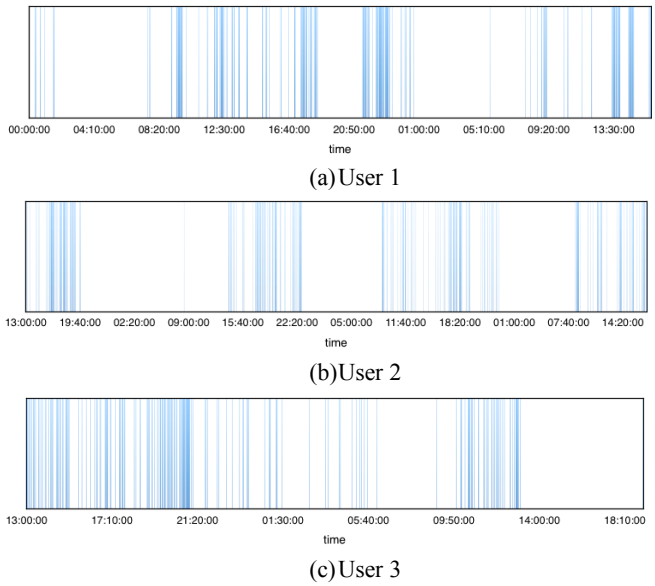

Figure 6: Distributions of sensing activities in different pedometer user traces.

In order to evaluate the energy optimization effects in a real environment, we installed the optimized pedometer to the smartphones of three different volunteers to collect real user traces. The app is modified to record the sensing rate level and callback counts. Figure 6 shows the distribution of sensing activities in the user traces. Each subfigure presents the trace for one user. The colored lines indicate the pedometer works at FASTEST rate level during the corresponding time period, and the blank indicate the pedometer works at the lower rate level (to make it simple, we do not distinguish the different lower rate levels in the figures.). We can see that the accelerometer samples at the lower rate mostly, while in the original app, the meter keeps running at the highest rate.

We calculate the total time of the pedometer working at each rate level, and recorded the total number of sampling counts, which are shown in Table 2. We can see that we are able to put the sensing frequency at the lowest level in over 90% of time duration.

We then calculated the energy consumption using the power numbers as shown in Figure 5, the energy comparison result is shown in Table 3. We are able to reduce the number of sensing actions by over 90%, while improve the whole device energy by more than 50% for all users.

5. Concluding Remarks

This paper proposes an instrumentation-based approach to fix sensor data underutilization in a mobile app automatically. With the help of an instrumentation framework, we are able to inject sensing policies into a mobile app, thus converting sensing mobile apps into energy-aware apps that can adjust sensing frequencies adaptively according to the current context.

User	Total time (s)	Time(s) (FASTEST)	Time(s) (GAME)	Time(s) (UI)	Time(s) (NORMAL)	Sampling count
1	131923	3475	1209	1306	125933	1410737
2	130136	5535	825	875	122901	1102359
3	57865	213	454	829	56369	333883

Table 2: Sensing statistics of pedometers for different users.

User	Total time (s)	Time(s) (FASTEST)	Time(s) (GAME)	Time(s) (UI)	Time(s) (NORMAL)	Sampling count
1	131923	3475	1209	1306	125933	1410737
2	130136	5535	825	875	122901	1102359
3	57865	213	454	829	56369	333883

Table 3: Sampling counts and energy saving for different users.

Through experiments, we demonstrate that it is effective and practical to instrument mobile pedometer apps to save energy automatically. We plan to explore more sensing policies and investigate the proposed techniques further to make it work for a wide range of mobile sensing apps.

Acknowledgments

This work is partly supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A203, the National Basic Research Program of China (973) under Grant No. 2011CB302604, and the National Natural Science Foundation of China under Grant No.61421091, 61370020，61103026.

References

1. Abhinav Pathak, Y Charlie Hu, and Ming Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices", in HotNet. ACM, 2011, p. 5.
2. Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel PMidkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps", in MobiSys. ACM, 2012, pp. 267-280.
3. Lide Zhang, Mark S Gordon, Robert P Dick, Z Morley Mao, Peter Dinda, and Lei Yang, "Adel: An automatic detector of energy leaks for smartphone applications", in CODES. ACM, 2012, pp. 363-372.
4. Yepang Liu, Chang Xu, and SC Cheung, "Where has my battery gone? finding sensor related energy black holes in smartphone applications", in PerCom. IEEE, 2013, pp. 2-10.
5. Y. Liu, C. Xu, S. Cheung, and J. Lu, "Greendroid: Automated diagnosis of energy ineffciency for smartphone applications", IEEE Trans on Software Engineering, 2014.
6. Heitor S Ramos and Nissanka B Priyantha, "LEAP: A Low Energy Assisted GPS for Trajectory-Based Services", in Proc. of UbiComp'11. 2011, pp. 335-344, ACM.
7. Bodhi Priyantha, Dimitrios Lymberopoulos, and Jie Liu, "LittleRock: Enabling Energy-Effcient Continuous Sensing on Mobile Phones", IEEE Pervasive Computing, vol. 10, no. 2, pp. 12-15, 2011.
8. Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo, "Dsp.ear: Leveraging co-processor support for continuous audio sensing on smartphones", in SenSys, 2014, pp. 295-309, ACM.
9. Donnie H Kim, Younghun Kim, Deborah Estrin, and Mani B Srivastava, "SensLoc: Sensing Everyday Places and Paths using Less Energy", in SenSys, 2010, pp. 43-56.
10. Nirmalya Roy, Archan Misra, Christine Julien, Sajal K. Das, and Jit Biswas, "An energy-effcient quality adaptive framework for multi-modal sensor context recognition", in PerCom. 2011, pp. 63-73, IEEE.
11. Suman Nath, "ACE: exploiting correlation for energy-effcient and continuous context sensing", in Proc. of MobiSys'12. 2012, pp. 29-42, ACM.
12. Kartik Sankaran, Minhui Zhu, Xiang Fa Guo, Akkihebbal L. Ananda, Mun Choon Chan, and Li-Shiuan Peh, "Using mobile phone barometer for low-power transportation context detection", in SenSys, 2014, pp. 191-205, ACM.
13. Kaisen Lin, Aman Kansal, Dimitrios Lymberopoulos, and Feng Zhao, "Energy-accuracy trade-off for continuous mobile device location", in MobiSys'10, 2010, pp. 285-298.
14. Jeongyeup Paek, Joongheon Kim, and Ramesh Govindan, "Energy-effcient rate-adaptive GPS-based positioning for smartphones", in Proc. of MobiSys'10, 2010, pp. 299-314.
15. Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh, "Improving energy effciency of location sensing on smartphones", in MobiSys. 2010, pp. 315-330, ACM.
16. Kent W. Nixon, Xiang Chen, Hucheng Zhou, Yunxin Liu, and Yiran Chen, "Mobile gpu power consumption reduction via dynamic resolution and frame rate scaling", in 6th Workshop on Power-Aware Computing and Systems (HotPower 14), Broomfield, CO, Oct. 2014, USENIX Association.
17. Robert LiKamWa, Bodhi Priyantha, Matthai Philipose, Lin Zhong, and Paramvir Bahl, "Energy characterization and optimization of image sensing toward continuous mobile vision", in MobiSys. 2013, pp. 69-82, ACM.
18. Aman Kansal, Scott Saponas, AJ Brush, Kathryn S McKinley, Todd Mytkowicz, and Ryder Ziola, "The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy effciency for continuous mobile context sensing", in OOPSLA. ACM, 2013, pp. 661-676.