User Interaction-based Profiling System for Android Application Tuning

Seokjun Lee, Chanmin Yoon, Hojung Cha
Department of Computer Science, Yonsei University, Seoul, Korea
{seokjunlee, cmyoon, hjcha}@yonsei.ac.kr

ABSTRACT
Quality improvement in mobile applications should be based on the consideration of several factors, such as users' diversity in spatio-temporal usage, as well as the device's resource usage, including battery life. Although application tuning should consider this practical issue, it is difficult to ensure the success of this process during the development stage due to the lack of information about application usage. This paper proposes a user interaction-based profiling system to overcome the limitations of development-level application debugging. In our system, the analysis of both device behavior and energy consumption is possible with fine-grained process-level application monitoring. By providing fine-grained information, including user interaction, system behavior, and power consumption, our system provides meaningful analysis for application tuning. The proposed method does not require the source code of the application and uses a web-based framework so that users can easily provide their usage data. Our case study with a few popular applications demonstrates that the proposed system is practical and useful for application tuning.

Author Keywords
Mobile Application; Profiling; Tuning; Debugging; Web-based Framework; User Interaction

ACM Classification Keywords
K.6.3 Management of Computing and Information System: Software Management; K.8.3 Management/Maintenance

INTRODUCTION

The development of mobile applications has accelerated as a result of the growth of the mobile ecosystem. The application market, such as Google's Google Play and Apple's AppStore have provided solid channels to distribute mobile applications. The number of application downloads has exponentially increased, but so has the number of complaints regarding the quality of mobile applications. According to research into mobile application consumers [1], more than half of all smartphone users have experienced application problems, such as poor user interface (UI), poor performance, inefficiency in battery life, unexpected application behavior, malfunctions in certain conditions, and programmatic issues such as crashes or freezes. Therefore, a good mobile application should assure good performance as well as energy efficiency, and it should not contain any erroneous behavior.

UbiComp '14, September 13 – 17, 2014, Seattle, WA, USA

http://dx.doi.org/10.1145/2632048.2636091

To improve the application quality, the developer should carefully consider functionality validation as well as its efficiency. Conventional programming tools available for developers include Dalvik Debug Monitor Server (DDMS) [2], Traceview [3], and iOS Xcode [4]. These tools are typically source-level debuggers and focus on the analysis of errors and performance, which has the following limitations. First, the actual environment of application running is hardly considered. Only a handful of usage cases are tested during development, and it is difficult to cover the diverse environments for application running. For example, depending on the location of the mobile user, the network status such as signal strength or network type influences the device behavior. Hence, tools for monitoring the application's behavior in real environments should be essential for the developer. Second, there is a limit in application testing at the development stage. Many tools used in application development and validation require a solid test plan, focusing on the verification of each functionality as well as performance. It is not possible to test the application thoroughly due to the exceptional runtime environment factors that are not expected in the development stage. In other words, tools for tests in real environments are necessary for meaningful application tuning.

Recently, several profiling tools [6-11, 23-29] for monitoring device behavior or application behavior have been proposed. They monitor some of the intrinsic issues that are difficult to analyze for other problems than those identifiable on the source code. Such issues include hardware component overuse, inefficient utilization of energy due to lack of understanding of power management by the operating system (OS), and exceptions invoked by resident processes or a change in the network environment. These profilers generally provide insufficient hints for understanding of the root-cause of problems that is essential for application tuning.

In this paper, we propose a profiling system for Android application tuning that overcomes the limitations of development-level program debugging, and which subsequently provides significant hints for application tuning. The proposed system aims to provide application developers

with information on users' usage activities, including both software and hardware operations, according to the user's specific actions and the energy implications. For instance, a newly developed application is tested in real environments; then the results are transmitted to server where the test results are analyzed by the developers for fine turning of the application at hand. This process certainly helps the developer find suspicious behavior, understand its root-cause, and determine whether it is indeed a problem. Hence, an application can be tuned more effectively by detecting and analyzing the potential problems in the implemented application. The key characteristics of the proposed system are as follows:

- The system does not require source code for profiling. Source-level profiling and debugging are restricted to the target application only; therefore, it is difficult to analyze the overall usage of the device and other applications whose source codes are not available. Our system monitors the overall device behavior, including a target application at runtime in the kernel and the Android framework-level.
- Our profiling system is web-based. For mobile application tuning to cover a wide range of usage cases in various environments, the profiling method should provide, without external tools, a handy interface to analyze profiling data, regardless of time and place. The proposed system performs profiling on the mobile device and transmits the collected data via the web interface. This allows the profiling of various usage cases and hence enables effective analysis.
- Our system provides a fine-grained level of application power tracing. Tracing the device's power consumption is essential for validating the application's energy efficiency. Our system uses AppScope [9] to trace the energy expenditure per-process and per-hardware component in detail without requiring an external power measuring device.
- Analyzing detailed user interaction is possible with our system. Profile data such as hardware usage or the amount of energy consumption are insufficient to find or replay the problem. The proposed system monitors and matches the user's detailed actions with the device behavior and helps the developers find useful clues to scrutinize outstanding issues.

REQUIREMENTS FOR MOBILE APPLICATION TUNING

Mobile applications share a very different set of characteristics compared to traditional desktop applications. For the successful development of mobile applications, the following aspects need to be considered in addition to conventional tuning activities.

There are many environmental factors that influence device behavior. The quality of an application is therefore influenced by diverse factors such as the GPS signal, the user's usage pattern, and so on. For example, the GPS signal changes while the user moves around and hence may cause an unexpected problem. Reflecting various environmental factors has the benefit of preventing unexpected issues that may arise after application deployment.

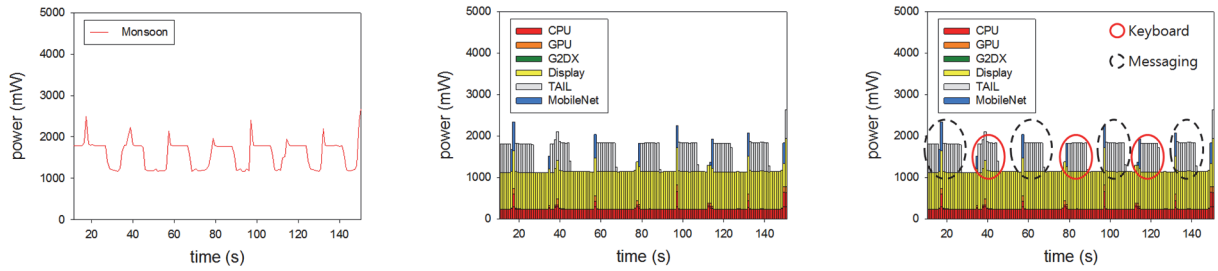
Mobile applications closely deal with frequent interactions between the user and the device, and reflect the user's behavior to the application operation. In other words, user interaction is a vital clue to find the root-cause of the problem, which is essential to improve the application. Therefore, the interaction between the user and the device should be carefully considered in the application tuning process.

The battery in the mobile device is an especially critical resource. Battery consumption is closely linked to the amount of hardware component usage and to system performance. Hence, the power consumption characteristics of each hardware component should be carefully considered during the application development. The trade-off between performance and energy needs to be carefully analyzed, and any possibility for battery inefficiency should be addressed.

However, considering all these factors in the development stage is very difficult in practice. Tests in the final stage of development are normally restricted to the functionality aspect with only limited resources; as a result, sophisticated analysis of problems that would require an in-depth understanding is hardly possible. Even after the application is deployed, it is by no means an easy task to manage and improve the application quality. Although the application marketplaces feature an easy approach to user applications, they lack support for problem reporting and evaluation. Currently, most application rating systems rely only on comment-based user feedback, and thus developers and testers find it hard to perceive problems. For practical application tuning, a good degree of field test is necessary in real environments, and the outcome should be effectively fed back to the developer for further enhancement. Even with the instrumented code for debugging, the verified points are restricted to whether the application operates as planned under specified conditions, whether the UI standard is followed throughout, whether legal requirements for distribution are met, and whether the application is resistant to ill-intended user control. In reality, it is not possible to check for various user environments, operations of hardware components, hardware overuse, energy overuse, or semantically wrong operations. Hence, a tool is needed that can collect fine-grained profile data, even without source code, and which helps developers easily find the root-cause of problems.

Motivational Example

In order to improve the quality of mobile applications, the factors that influence application quality should be identified and analyzed accordingly. Errors in the source code can easily be discovered during the debugging stage, but the more complicated problems, such as the inefficiency of resource utilization or inappropriate operations caused by an unexpected environment, are not easily detected during



(a) Monsoon power trace **(b) AppScope power trace** **(c) AppScope with annotation**

Figure 1. Snapshot of component-level energy consumption for the messaging application "Kakao Talk."

application execution. Thus, locating and improving such factors is a challenging task.

For example, measuring power consumption using an external tool enables producing an accurate number, but the limited data makes it hard to perform meaningful application tuning. Meanwhile, fine-grained power estimation tools, such as AppScope [9], eProf [10], or that of Li, et al. [11], allow for measuring the power consumption of the device per-hardware component and even per-function call. This information is more detailed than that acquired with external power gauging tools. However, knowing the power consumption behavior does not necessarily enable a developer to locate and identify the origin and source of inappropriate device behavior.

Figure 1 shows the results of measuring power consumption for a messenger application called "Kakao Talk," which was tested on the Samsung Galaxy S3 (SHV-E210S) with an external tool and AppScope [9]. The result with an external tool offers only the total amount of the power consumption on the device. The per-hardware component power consumption chart in Figure 1(b) shows that a mobile network that consumes a significant amount of power may be wasting battery power unnecessarily. However, based on the information in Figure 1(b) alone, it is not possible to discern the reasons for the mobile traffic; therefore, we cannot determine whether this can be considered as inefficient power consumption. Meanwhile, Figure 1(c) is an annotation of how the experiment illustrated in Figure 1(b) was conducted. Red circle points signify that the user touched the message box to make the soft keyboard appear. The black circles represent where the messages were sent. It is evident that the tail power activities marked with the red circles are the sources of energy inefficiency, since there is no reason for data transmission to occur. Even with no in-depth understanding of the characteristics of the mobile network, e.g., tail power consumption, energy inefficiency can easily be noticed, and the developer is given an opportunity to consider the possible causes as well as ways to address the inefficiency.

To summarize, an adequate process of application tuning should consider the following requirements. First, fine-grained energy monitoring needs to be possible. To observe the device behavior with the visibility of specified applications and consumed energy, detailed data on the component-level is necessary. Second, the root-cause of the observed device energy behavior must be identified, because this makes it possible to decide whether the observed device behavior is a problem, where the problematic behavior happens, and how such problems can be improved. In other words, detailed data on user interactions should be provided to find a root-cause of device behavior. Finally, the solution should make it easy to collect actual usage data. In the Google Android developer resource [12], it is stated that one of the most important methods of improving application quality is listening to users. Users use the applications under diverse conditions. Hence, easy data collection for a wide range of environments is very important in test process.

SYSTEM DESIGN AND IMPLEMENTATION

We now present the design and implementation of a user interaction-based profiling system to meet the requirements described above. Figure 2 shows the overview of the system. Users and developers initiate the profiling process to analyze and improve the application quality. In the mobile device, the system monitors hardware usage as well as user interaction and transfers this data to the server. The profiling data is analyzed and stored in the database. Developers may analyze the profiling data with various graphs and statistics via the web interface. Developers then tune the application and perform the profiling again to verify the result of tuning. Application quality may improve gradually through this cycle. Note that the proposed system is developed on Nexus 4 with a modification of Android 4.3 and the kernel.

Mobile Device Side

Our system monitors the kernel behavior to trace hardware usage and the framework behaviors to trace detailed user interaction as well as system behavior. Logging is handled by a monitoring service that runs in the background. The user controls the monitoring service with a control application, which manages and transfers the profiling data. A detailed description of the system is given below.

Kernel Level

Our system adopted AppScope [9] to trace fine-grained hardware usage. AppScope monitors diverse kernel behavior, including binder transaction, device driver, and Linux functions, to collect process-level usage of hardware

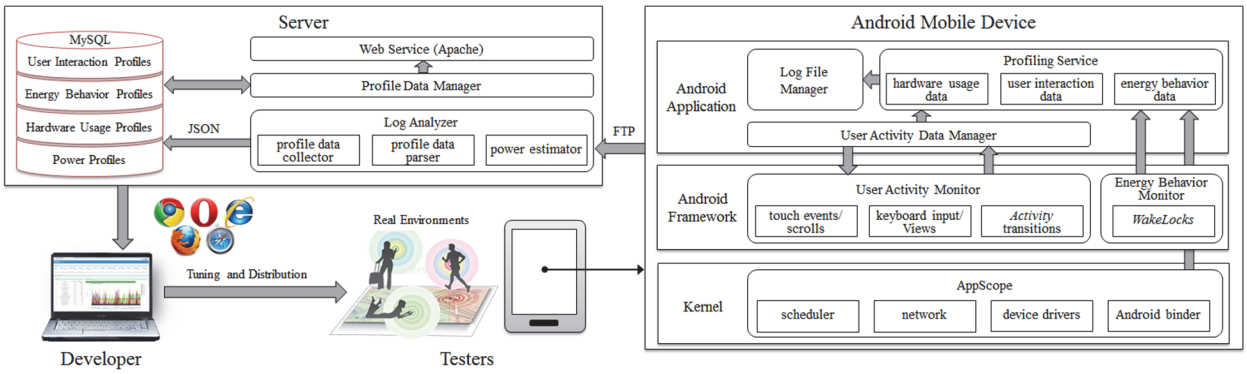


Figure 2. System overview.

components such as CPU, GPU, GPS, display, Wi-Fi, and mobile.

Framework Level

We investigated the Android framework to trace user interactions and system behavior in detail. *Activity* is a primary class to run the application and provide UI, which enables users to interact with the device. Therefore, *Activity* is a basic hint for identifying user action. In addition, a single *Activity* consists of various UI components, including View and Widget. Other actions, such as keyboard event, touch, and scroll, are also invoked by the user. Determining the exact user action that is a root-cause of device behavior requires a detailed monitoring of user interaction. The proposed system traces interactions, including keyboard activity, touch, scroll, click/long click, as well as *Activity* transition. Meanwhile, a common Android system behavior that influences resource efficiency is wakelock [6-8]. Our system traces the wakelock operation to analyze and improve its usage.

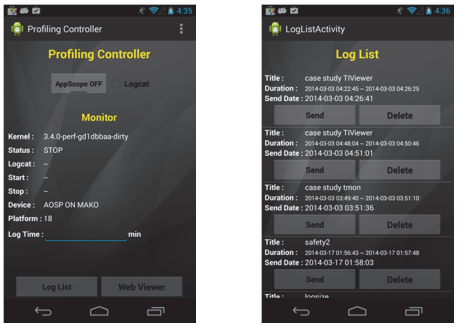
We modified Android 4.3 to implement the profiling system. First, we created an Android system service, "UserActivityService," to produce an interaction log and access to log data. Log management using the system service rather than a static class or global variable is required to retain the log over different program contexts. Since the system service is accessible by the service manager, we implemented a service manager, "UserActivityManager," which also delivers the interaction log to the application level. We inserted a logging method into some points in the framework where interactions occur. For example, the logging method is placed in the View class to trace click/long click events.

Application Level

At the application level, a background monitoring service merges interaction, wakelock, and power trace into a single log file. The service retrieves hardware usage from AppScope via the *proc* file system. It also uses UserActivityManager to obtain the interaction and wakelock trace from UserActivityService. We provide a control application for service control and log management. The application collects environmental information, such as device name, and transfers the profiling data to the server. Figure 3 shows the screenshot of the control application. The upper toggle button in Figure 3(a) starts/stops the logging service, and the "Log List" button in Figure 3(a) lists the collected log data shown in Figure 3(b). In addition to all-day background logging, this application enables testers to specify the characteristics of profiling data. For example, a tester may collect profiling data in a building where the GPS signal strength is weak, and name it "In Building." This feature enables developers to find the problems and focus on them more easily.

Time synchronization. Monitoring service retrieves data for every second. Since the elapsed time for one second is different in the kernel and in the framework, merging the log data requires time synchronization. We addressed this issue by accumulating each log both in the kernel and the framework until the monitoring service sends a request. After receiving the request, both logs are delivered to the monitoring service simultaneously. Through this method, the logging interval of both the kernel and the framework became the same.

Interacting object. Identifying an interacting object using only the class name and ID without the source code is difficult. To address this issue, we also collected the information on the overall layout of the device as well as the position of the interacting object. Our system retrieves all parent views of an interacting object recursively for the



(a) Main activity (b) Log list activity

Figure 3. Control application.

overall layout. With this method, developers are able to identify the interacting object intuitively.

Server Side

The transmitted log is transformed to form graphs and tables, and is then stored in the database. The system provides this data to the client via web interface. We implemented the server using Apache, PHP, and MySQL, while we implemented the client-side analysis tool using the Google chart Application Program Interface (API) [13].

Log Parser

The log parser processes the transmitted log into analyzable data. First, hardware usage is stored in the database. Then, the power trace is generated using the power model [14], which is stored in the database in advance. Meanwhile, interaction data is grouped by type and arranged over time. Finally, the system transforms the power and the interaction traces to JavaScript Standard Object Notation (JSON), which is used in the Google chart API [13].

Event-driven parsing. Periodic searching of a newly uploaded log causes resource waste and a delay between upload and parsing. We created a trigger *php* file for event-driven parsing. The control application accesses this page automatically after log transmission. As a result, the server is able to process the log immediately.

Profile Data Manager

The data manager handles the data request from a client and updates the data list. When the log is newly uploaded, its environmental information is added to the list. The data manager also retrieves JSON from the database and transfers it to the client corresponding to the user selection.

Low latency. According to logging time, the size of JSON may increase, and consequently latency at the client-side becomes longer. Hence, we generated multiple JSON for each process, rather than a single JSON, in order to reduce the size of each JSON. This minimized the transmission time and consequently achieved low latency.

User Interface

Figure 4 shows the overall user interface of the profiling system, and Figure 5 overviews the user interface of log probing, respectively. Our system is available at *http://165.132.122.93/ApplicationTuning/ProfilingSystem.html*. We synchronized the time axis of all trace data to analyze various information at a glance. The UI consists of the analysis page (A~F section) and the probing page (G~H section). The detailed description for each section is given below.

A. Log management. The system provides a profiling data list at the top part of the UI. Users can filter it depending on environment information such as device name and logging date. The functionalities, including list toggle and data erasure, are also provided for efficient analysis.

B. Fine-grained power trace. Device power trace is displayed on a stacked column graph to represent component-level power consumption as well as device behavior. Developers can analyze the total amount of power consumption and also the ratio and the amount of power consumption for individual component at a glance. Each component is displayed selectively by a filter. For example, the application developer for location service can focus only on the GPS and network components. This enables the developer to understand the device behavior and find potential problems more easily. The power trace of a single process ID (PID) or user ID (UID) is also presented when the user selects PID/UID from the table in section D. In addition, developers can freely zoom the graph in/out. These features greatly help the in-depth analysis of the energy characteristics of a single application.

C. Detailed trace on user interaction. A set of traces on user interactions is shown on a timeline. Detailed information is provided as a tooltip when the user hovers the mouse pointer. In the activity tooltip, both the package name and the *Activity* name are provided. In a view click, the ID set by the developer and class name are presented. This helps the developers find the interacting objects of their application. Even for an external application such as Google Play, E-mail, or Messenger, the developer can easily identify the interacting objects. Other interaction events, such as touch, slide, soft keyboard toggle, and typing, are also displayed along the timeline. The developer understands the user interactions by observing such information at a glance. Analysis of these interactions helps to find the exact user behavior and the root-cause of a potential problem.

D. WakeLock. As shown in Figure 4, the acquire/release time of wakelock is presented with the owner UID. Two UIDs that have acquired and released the wakelock are expressed together. Detailed information, such as flag and tag, is provided using the tooltip. This enables intuitive analysis and tuning on wakelock usage.

E. Energy consumption stats. The energy consumption statistics at the PID/UID level are illustrated by both a table and a pie chart. The power trace and information about a single PID/UID are presented according to user selection. This enables an in-depth analysis of the battery consumption and device behavior.

F. View emulator. Our system provides the overall layout of the device and the position of the interacting object in order to identify the user activity more intuitively. Layout is represented by black lines, and an interacting object is highlighted using red lines. The developer can visually trace user interactions by a simple series of clicks. This greatly reduces the analysis cost. We used the canvas of HTML5 to draw the view emulator dynamically.

G. Probing criteria. Our system provides a probing functionality to help the developers navigate the voluminous traces and hopefully spot the suspicious behaviors. As shown in Figure 5, the developers set probing criteria such as logging date, target *Activity*, target device, and target range

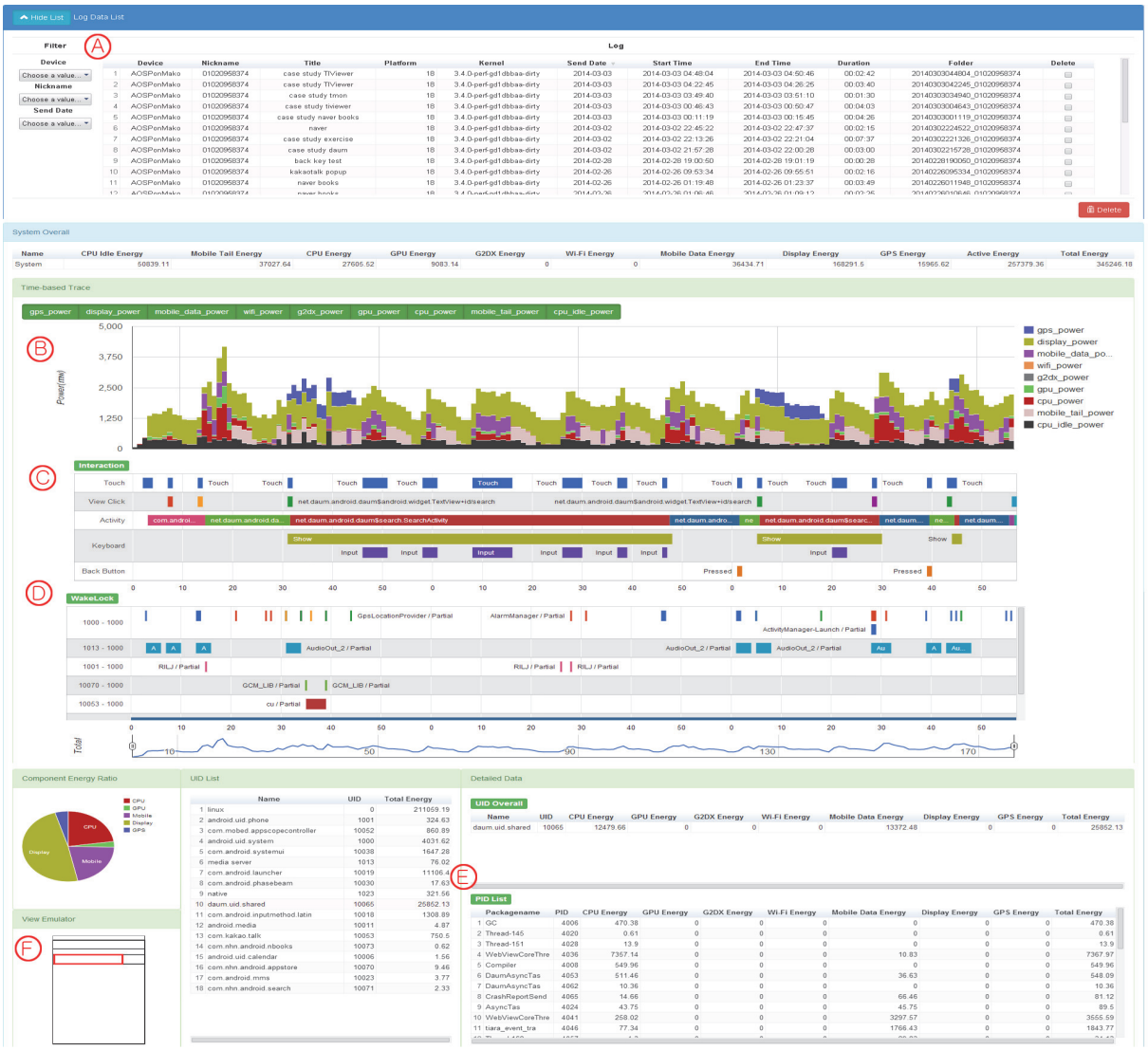


Figure 4. Overall user interface of interaction-centric device behavior analysis system.

of power consumption for each component. With these options, the developers can narrow down the scope of suspicious behavior.

H. Probing list. After a probing request, the system searches for logs to meet the criteria, then returns the probing list that consists of logging information, power trace, and interaction trace for each log. As shown in Figure 5, the part of the power trace that meets the probing criteria is emphasized with high opacity in color. The probing list now enables the developers to investigate the suspicious behaviors effectively over large profiling data.

EVALUATION

We conducted a series of case studies with real applications in Google Play to validate the usefulness of our system. Evaluation of overhead and accuracy of power estimation was also conducted.

Case Study

We analyzed the profiling data of five widely used Android applications, as shown in Table 1.

An application called "Daum," which is an Android version of the web portal, "Daum." We describe the detailed procedure for application tuning with our system. Our system is operated over three phases: i) probing a suspicious behavior, ii) analyzing the root-cause of behavior, and iii) finding a tuning point. We now describe each phase in detail with the "Daum" application.

i) Probing suspicious behavior. Based on the characteristics of the application, the developers narrow down the scope of suspicious behavior. Based on the probing criteria, they obtain a probing list, as presented in Figure 6. The developers investigate the list of logs and find suspicious behaviors. In the case of "Daum," we observed the GPS activation during logging and hence, investigated the GPS-related behavior. In this case, we set the probing criteria as follows: *Activity* name:

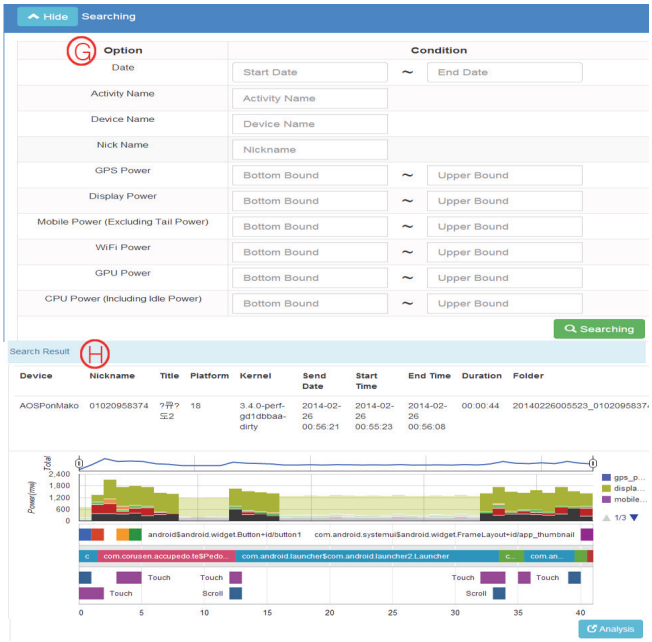


Figure 5. User interface of log probing.

"Daum" and GPS power > 100. Figure 6 (a) shows one of the probing results where the GPS-related behavior is presented with high opacity in display color. Since the third instance of GPS behavior looked abnormal, we regarded it as a suspicious behavior.

ii) Analyzing root-cause of behavior. The developers now analyze the suspicious behavior in detail *at the analysis page*. With both the user interaction trace and the power trace, the developers find the root-cause of suspicious behavior. According to the interaction and the power trace in Figure 6 (b), GPS was activated whenever the user entered the searching *Activity*. This enabled us to infer that the GPS was activated for location-based searching.

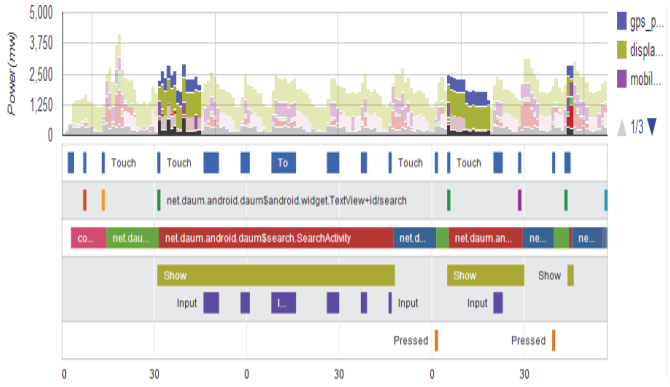
iii) Finding application tuning point. Finally, the developers find an application tuning point based on the root-cause of the suspicious behavior. They determine the tuning direction, considering various factors such as their initial intention,

Name	Category	Download	Rating	Reviewer
Daum	Communication	+10,000,000	4.2	49,592
Fashion Muscle Exercise	Health & Fitness	+500,000	4.4	905
Ticket Monster	Shopping	+5,000,000	4.0	13,460
TIViewer	Media & Moview	+1,000,000	4.4	8,647
Naver Books	Books & Reference	+1,000,000	3.8	10,663
Smart Safe Homecomming	Life Style	+100,000	3.9	513

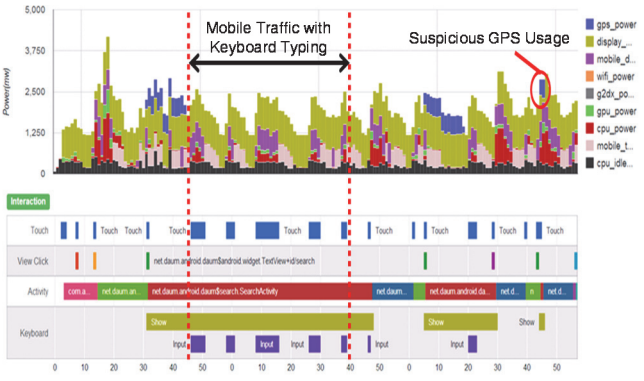
Table 1. Real application for case study.

tradeoffs between overhead and performance, unexpected environment, and others. In this case study, GPS was activated for only two seconds at the third searching activity. This may be an inappropriate GPS usage, because the GPS stopped before receiving the location information. If location is essential for searching, the developer may modify the application to keep GPS activated until the location is acquired properly.

- **Root-cause of mobile traffic.** We analyzed another suspicious behavior in terms of mobile network. In Figure 6 (b), the section on keyboard typing is indicated by a red dotted line. According to the power trace, mobile traffic occurred, and energy consumption is consequently observed during this section. However, analyzing only the power consumption is not sufficient to find the exact cause of the mobile component behavior. The developer is unable to determine if the observed behavior is appropriate. On the other hand, analysis with information on user interaction is certainly helpful to infer the root-cause. For example, keyboard event, click event, and *Activity* transition help us to find that the user typed a keyword in the search box at SearchActivity. In addition, analysis with both the interaction trace and the power trace reveals that the keyboard typing and the mobile traffic occurred at the same time. This means that typing in the search box has led to mobile traffic. Since keyboard typing is identified as a root-cause of mobile traffic, the developer cannot tell whether this mobile traffic is intended or not. Unintended traffic means that energy is wasted, and hence the application should be improved by adequate debugging. By contrast, in the case of intended



(a) Probing result



(b) Detailed analysis at analysis page

Figure 6. Profiling data of "Daum" application when the user performs searching activity.

traffic such as auto-completion, the developer is able to analyze the overhead of such traffic. Furthermore, the developer could adjust the balance between battery consumption and user convenience by providing an On/Off setting for auto-completion.

Social commerce application, "Ticket Monster." This program finds coupons around the current location or region given by the user. Figure 7 shows the profiling data when the user was searching the coupons. We derive a finding by analyzing the profiling data.

- **Tradeoff between GPS usage and user experience.** The interaction and power trace illustrated in the last section of Figure 7 indicate that GPS is activated, even though the current location is no longer needed. Regarding energy efficiency, this means that GPS is utilized redundantly, and energy is wasted. Users are, however, expected to be notified of location-based coupons instantly by GPS being activated. The developer should therefore consider the tradeoff between the GPS usage and the user experience for better application service. For example, the developer would be able to enhance the energy efficiency by deactivating GPS when the user searches for coupons around a fixed location.

Exercise guide application, "Fashion Muscle Exercise." This application informs the user of various exercise methods. Users often read an exercise guide and practice the exercises without any interaction. Figure 8 illustrates the profiling data when the user reads an exercise guide. We observe the following finding.

- **Redundant mobile traffic.** According to the power and interaction traces, mobile traffic occurred even though there was no user interaction; consequently, energy waste also occurred. In a mobile network, a small amount of traffic often causes large energy consumption, because of the tail state. Periodic traffic with short intervals may therefore lead to significant battery consumption. Based on the profiling data, the developer and the user may notice that the mobile traffic is caused by a background job. Furthermore, by considering the purpose of the application, the developer can determine that this traffic is redundant and hence can improve the application quality by merging the traffic.

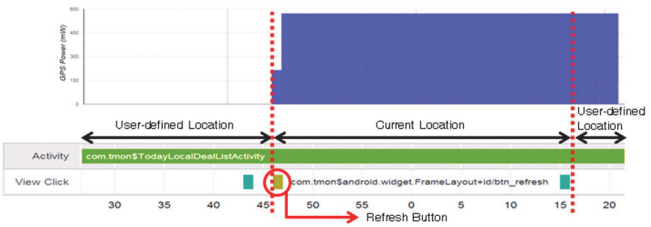


Figure 7. Snapshot of profiling data for "Ticket Monster" app when the user found social coupons depending on the location.

E-book application, "Naver Books," which is reported as a wakelock anomaly app [8]. Naver Books provides e-book purchasing and reading. Figure 10 shows the wakelock and interaction trace when the user escaped to the home launcher after reading activity. We know that the application acquired a *SCREEN BRIGHT* wakelock to prevent the screen from turning off while reading an e-book. The wakelock, however, was not released even though the user moved to the home launcher. This causes the screen to remain awake unnecessarily, and thus results in significant energy waste. With this information, the developer can debug the misused wakelock operation.

Text and image viewer application, "TIViewer." TIViewer is a viewer for text or image. Figure 9 shows the profiling data when the user reads text. The application "Kakao Talk," in particular, was executed for a while in the middle of reading. We suggest the following points for improvement.

- ***Activity* transition against the Android navigation principle.** Google provides a set of basic navigation principles [12]. They suggest that navigation with the back button should explore the screens in the reverse direction of the time order rather than following the app hierarchy. The *Activity* trace in Figure 9 shows that the home launcher, which is highlighted as red, was executed instead of TIViewer when the user pressed the back button at the "Kakao Talk Main" *Activity*. This does not coincide with the navigation principle of Google. Moreover, the user should perform a redundant action, such as opening the application drawer or recent activity list, in order to return to TIViewer. The developer can now tune the

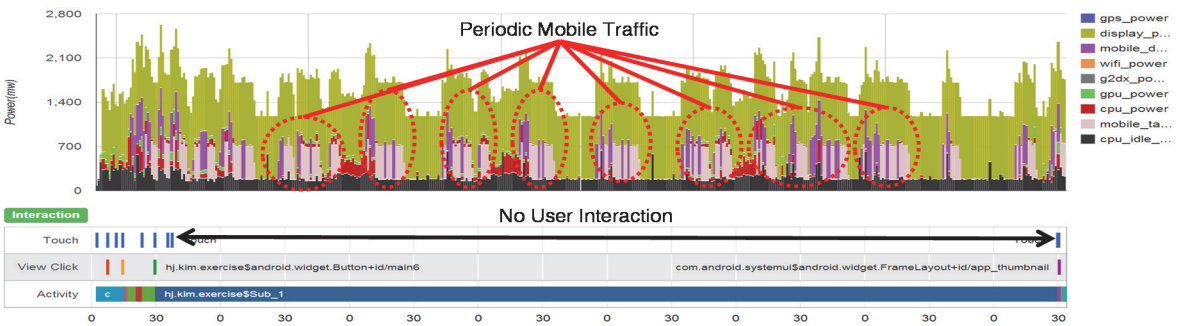


Figure 8. Profiling data of "Fashion Muscle Exercise" application.

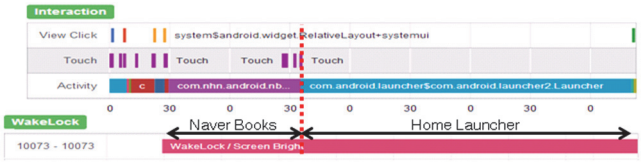


Figure 10. Profiling data of "Naver Books" application.

application by providing an appropriate navigation for the user.

Application for safe homecoming, "Smart Safe Homecoming." This application provides a location-based service for safe homecoming. Figure 11 demonstrates that a user tried to start the service but failed, due to weak GPS signals. The service was merely waiting for location information without responding to the user interaction event for the back button. We observed the following facts.

- **Problem with unexpected environments.** The weak GPS signal led to the service failure from the beginning. Thus, user inconvenience as well as significant energy consumption occurred, as shown in Figure 11. At the red box point, the service did not terminate even though the user pressed the back button. In addition, the amount of GPS energy consumption is observed until the end of the application. The environment where the GPS signal is weak is often observed under certain situations. The developer may have not considered this case because it did not appear in the development stage. Through the real usage data provided by an actual user, developers now easily analyze the various environments as well as the potential problems in their program.
- **Necessity of user interaction.** As shown in this case study, detailed user interaction is essential to find a problem, if any. If user interaction information such as back button or *Activity* were not provided, developers would not be able to detect any problem from this profiling data. With the power graph only, developers cannot determine whether the GPS usage is correct or whether the back button navigation is working properly.

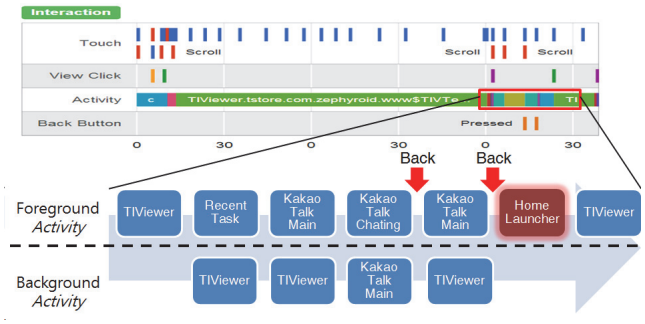


Figure 9. Profiling data of "TIViewer" application.

Overhead

We measured the CPU utilization, the power consumption, and the interaction delay for our system in order to assess the overhead. We conducted experiments on Nexus 4 using both the web browser and the home launcher. We repeated each test 10 times for fair evaluation. The CPU utilization is measured using the *proc* file system, power is gauged by Monsoon [15], and latency is estimated by calculating the delay to launch activity. Compared to a native system, the CPU overhead was 3%, the power overhead was 13.8 mW, and the latency overhead was almost zero. The result shows that the cost of our system is negligible. Meanwhile, the log size after profiling for one hour was about 2 MB, which can be sent by Wi-Fi within a few seconds. Note that our system can further minimize the transmission overhead by transferring the log only when Wi-Fi is available.

RELATED WORK

Active efforts have been made to improve application quality in various aspects. In the case of the Android platform, for example, tools such as DDMS [2] and Traceview [3] are available for code-level debugging.

Performance is the focal point for application quality improvement, since it has a major impact on user experience. In [16, 17], operation path is analyzed from the point when the user provides an input. Its corresponding output is shown on the display, and the longest path is shown to reduce response time. In [18], a performance bug in a mobile application is specifically defined, because performance issues on mobile devices are inherently different from desktop cases. With static analysis, such bugs are detected in

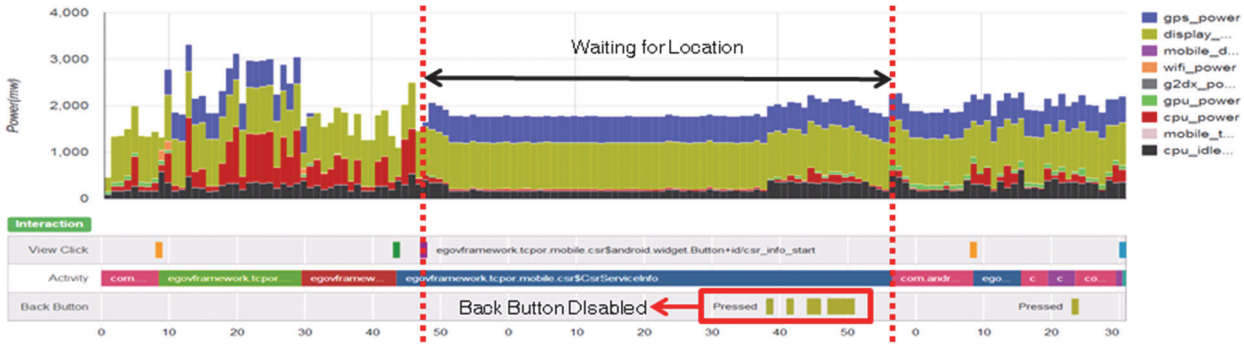


Figure 11. Profiling data of "Smart Safe Homecoming" application when the user tried to start the service.

real-world applications. Inconsistent user-perceived delay on server-based mobile applications was addressed by Ravindranath, et al. [19]. The scheme predicts the response time of the server-based application, which in turn is manipulated on the server for user-perceived delay control.

Others have made attempts to improve applications in terms of energy consumption. In [9-11], issues on overall energy measurement were addressed using an external gauging tool, such as Monsoon [15], or when using the state-of-discharge (SOD) information. When measuring only the overall energy consumption of the device, diagnosis of energy inefficiency is difficult. Therefore, AppScope [9] and eProf [10] proposed a per-hardware component and per-process energy monitoring systems. Energy was monitored for each line in [11] in order to improve battery efficiency. Other researchers took further steps to locate energy inefficiency beyond just merely monitoring energy consumption. In [6-8], energy bugs associated with incorrect usage of wakelock were sought for detection. In [20], energy inefficiency arising in suspend mode, depending on the behavior of the device driver, was explored. A detection method for applications and threads showing abnormal energy consumption was proposed in [22, 23].

In the software engineering field, graphic user interface (GUI) objects are automatically tested, and the conditions in which crashes occur are found and reported to the developer in order to make applications more reliable. Google Analytics [5] provides analysis of what activities are performed by users on the specified application. In particular, it shows the path of how and when in-app purchases occur. In [23], by using Robotium [25] based on an Android test project [24], improvements were made to overcome the limitation of conventional automatic analysis methods, where applications could be debugged only via emulation or only when the source was available. This method has widened test coverage by firing events for every GUI object on actual devices. In [26], it was assumed that testing for all scenario cases is inefficient and that the developer could specify the scope of testing so that testing could be conducted efficiently.

In [27], a method was proposed to analyze applications by considering the environmental variables such as user, device, and network; however, this study did not examine any direct aspect of application quality. A longitudinal observation was made for two real-world applications; nonetheless, the result was too coarse-grained to perform meaningful analysis. Falaki, et al. [28, 29] collected high-level information such as *Activity* and cell state, as well as low-level information, including CPU and memory usage. Falaki, et al. focused on the relation between variable factors rather than application tuning.

DISCUSSION

Our case study shows that a user interaction-based profiling system is promising for application tuning. First, our system can be utilized for field tests in real environments. Although the system needs kernel and framework modification, which makes a large-scale deployment of the system rather difficult, our system is certainly useful for field test by selected testers. Our system is also helpful for tests with fixed scenarios. With a test scenario fixed *a priori*, developers may miss the actual sequence of operations, which raises a problem. The system would help the developers thoroughly observe the actual activities.

Second, the proposed system is, in fact, a profiling tool, not a diagnosis tool. The system aims to help developers analyze log data and understand the device behavior. Although automatic problem detection is not our scope, combining our system with diagnosis systems such as AppInsight [16] or Carat [21] would certainly enhance the quality of application tuning. With the functional availability of performance diagnosis [16] and energy hog detection [21], for example, our system would provide in-depth analysis and useful information for application tuning.

Finally, we used popular applications for our case study to prove the necessity of user interaction and also to validate the practicality of our system. The applications were selected based on various criteria such as download counts, rating, and review counts, as shown in Table 1. With these applications, we showed that analysis of both power consumption and user interaction is definitely helpful for understanding root-cause of problems, and subsequently for improving application quality. This means that user interaction is important, and our system is indeed practical.

CONCLUSION

We proposed a user interaction-based profiling system for application tuning. With fine-grained energy monitoring, it was possible to observe the amount and characteristics of energy consumption by a specified application. User interaction data was also analyzed alongside such monitoring to help find the root-cause of energy inefficiency for application tuning. By providing a web-based framework with which users can easily provide real-time data, developers will acquire practical benefits.

In the future, we plan to devise methods to monitor the major components on the Android platform, expanding from user and device interactions to include service and alarm for more detailed and comprehensive analysis. We will go beyond merely showing monitoring results by offering probabilistic and statistical analysis results. With that information, the state and causes of energy inefficiency can be detected. Furthermore, we will also explore malware detection methods based on the relationship between interactions and device behaviors.

ACKNOWLEDGMENTS

This work was supported by a grant from the National Research Foundation of Korea (NRF), funded by the Korean government, Ministry of Education, Science and Technology under Grant (No.2013-027363).

REFERENCES

1. Mobile Apps: What Consumers Really Need and Want http://offers2.compuware.com/rs/compuware/images/Mobile_App_Survey_Report.pdf.
2. Android Dalvik Debug Monitor Server (DDMS). http://developer.android.com/tools/debugging/ddms.html.
3. Android Traceview http://developer.android.com/tools/debugging/debugging-tracing.html#traceviewLayout.
4. iOS Xcode https://developer.apple.com/library/ios/documentation/ToolsLanguages/Conceptual/Xcode_Overview/DebugYourApp/DebugYourApp.html.
5. Google Analytics http://www.google.com/analytics/.
6. A. Pathak, A. Jindal, Y. Charlie Hu, and S. P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of MobiSys '12*, ACM Press (2012), pp. 267-280.
7. A. Pathak, Y. Charlie Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proc. of HotNets-X '11*, ACM Press (2011), pp. 1-6.
8. K. Kim and H. Cha. WakeScope: Runtime WakeLock anomaly management scheme for Android platform. In *Proc. of EMSOFT '13*, IEEE (2013), p. 10.
9. C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Proc. of USENIX ATC '12, USENIX* (2012), pp. 36-36.
10. A. Pathak, Y. Charlie Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proc. of EuroSys '12*, ACM Press (2012), pp. 29-42.
11. D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for Android applications. In *Proc. of ISSTA '13*, ACM Press (2013), pp. 78-89.
12. Android developer resource http://developer.android.com/index.html.
13. Google chart API https://developers.google.com/chart/.
14. C. Yoon, G. Ryu, N. Jung, and H. Cha, Accurate power modeling for modern mobile application processors, Yonsei University, Tech. Rep., 2014. MOBED-TR-2014-1.
15. Monsoon http://www.msoon.com/LabEquipment/PowerMonitor/
16. L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proc. of OSDI '12*, USENIX (2102), pp. 107-120.
17. L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: Event-based tracing to measure mobile application and platform performance. In *CODES+ ISSS '13*, IEEE (2013), pp. 1-10.
18. Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proc. of ICSE '14*, 2014.
19. L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proc. of SOSP '13*, ACM Press (2013), pp. 85-100.
20. A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. Hypnos: Understanding and treating sleep conflicts in smartphones. In *Proc. of EuroSys '13*, ACM Press (2013), pp. 253-266.
21. A.J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proc. of Sensys '13*, ACM Press (2013), p. 14.
22. 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. of NSDI '13*, USENIX (2013), pp. 57-70.
23. T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proc. of OOPSLA '13*, ACM Press (2013), pp. 641-660.
24. Android testing technique http://developer.android.com/tools/testing/index.html.
25. Robotium https://code.google.com/p/robotium/.
26. S. Hao, D. Li, W. G.J. Halfond, and R. Govindan. SIF: a selective instrumentation framework for mobile applications. In *Proc. of MobiSys '13*, ACM Press (2013), pp. 167-180.
27. A. Patro, S. Rayanchu, M. Griepentrog, Y. Ma, and S. Banerjee. Capturing mobile experience in the wild: A tale of two apps. In *Proc. of CoNEXT '13*, ACM Press (2013), pp. 199-210.
28. H. Falaki, R. Mahajan, and D. Estrin. SystemSens: A tool for monitoring usage in smartphone research deployments. In *Proc. of MobiArch '11*, ACM Press (2011), pp. 25-30.
29. H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proc. of MobiSys '10*, ACM Press (2010), pp. 179-194