EnTrack: A System Facility for Analyzing Energy Consumption of Android System Services

Seokjun Lee, Wonwoo Jung, Yohan Chon, Hojung Cha
Department of Computer Science, Yonsei University, Seoul, Korea
{sjlee,wwjung,yohan,hjcha}@cs.yonsei.ac.kr

ABSTRACT
Energy accounting is an essential requirement for optimizing energy consumption on mobile devices. State-of-the-art approaches consider application processes and threads as the sole components of energy consumption. In this framework, the energy consumption of system services is unclear and has not been comprehensively studied. In this paper, we suggest that the energy consumption of system services should be investigated to understand the behavior of applications. We propose a fine-grained energy tracing scheme, EnTrack, to enhance the accuracy of energy tracing by identifying and incorporating the energy portions consumed by system services. We implemented EnTrack on the Android platform and validated its functionality and usefulness. In addition, practical usage cases of EnTrack, which uses it as an energy behavior analysis tool, were introduced. The case studies demonstrated that EnTrack enables an understanding of fine-grained energy consumption, especially in system services, which have previously been concealed.

Author Keywords
Energy consumption tracing; energy optimization; mobile systems

ACM Classification Keywords
C.4.8. Performance

INTRODUCTION
Energy efficiency is a critical issue for mobile devices. For the energy-efficient operation of devices, energy accounting is an essential requirement for understanding software units such as applications and related system software. When given the energy information for each application, energy behavior is intuitively understood by both users and application developers. Users are thus guided toward energy-aware use of their devices [1-5], and application developers are advised to consider energy efficiency in their code development [6-8]. The information is also critical for the development of energy-aware mobile operating systems that manage applications' energy usage efficiently [9-14].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UbiComp '15, September 7–11, 2015, Osaka, Japan.
Copyright 2015 © ACM 978-1-4503-3574-4/15/09…$15.00.
http://dx.doi.org/10.1145/2750858.2807531

Previous studies have focused predominantly on energy accounting processes or threads rather than the applications themselves [14-17]. The energy consumption of each application is usually considered as the summation of the energy consumption of the constituent processes (e.g., the processes and threads with the same UID). However, such approaches often do not represent the exact energy consumption of applications due to the complicated features of modern mobile platforms.

In modern platforms, *system services* that support diverse types of hardware requests by applications operate in the middleware between the applications and the operating system. In this framework, an energy accounting scheme that calculates the energy consumption of applications based on the processes/threads groups is unlikely to be accurate since a considerable portion of the energy consumed by system services is used on behalf of applications. Figure 1(a) illustrates the processes/thread-based energy estimation approach. The approach measures the energy consumption of each process; hence the relationship between the application and system service is unclear. In addition, the energy consumption of a system service is concealed since a process can have many system services. For example, the representative Android process *system_server* runs approximately 40 services, such as *ActivityManager*, *LocationManager*, and so on. The fine-grained operations and energy consumption of the system services are therefore a black box to application developers in situations where the applications are tightly linked with the various system services that are running.

This paper introduces EnTrack, a software facility that accurately identifies the fine-grained energy consumption of system services. Figure 1(b) shows the conceptual view of EnTrack. Compared to the process/thread-based approach [14-16] including our previous work of AppScope [17], EnTrack provides a new set of information: (1) the relationship between applications and services (i.e., which application invokes which functions in which services), (2) the fine-grained energy consumption of system services according to function unit (e.g., 100mJ at *android.media.*

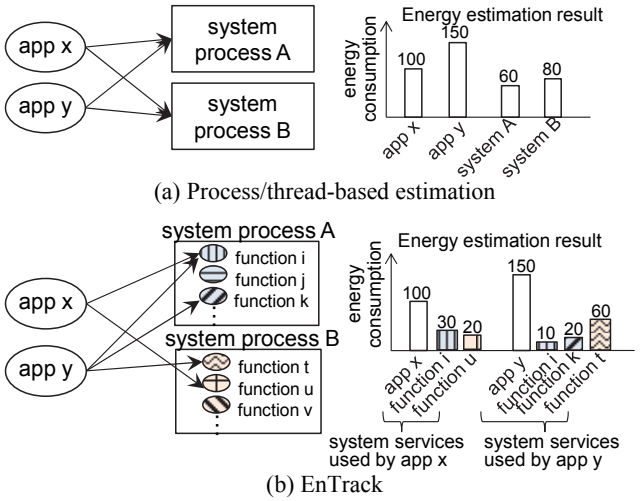


Figure 1: Conceptual view and energy estimation result of (a) the process/thread-based estimation approach [15-18] and (b) the proposed system. The edge indicates that the application calls the system service.

IAudioService.adjustVolume), and (3) the energy consumption of applications linked with system services. EnTrack provides these functionalities online without modifying the application source. With EnTrack, the following development questions can be answered:

- How much energy does our application, including the system services, consume?
- What is the root cause of abnormal energy consumption, if such consumption exists? Is it the mobile application we implemented or the system service we used with the application?
- How much energy do the system services consume? Are there any energy leaks after a system update?
- How can we reduce the energy consumption of our mobile application when using the system services?

We evaluate the functionality of EnTrack with several applications. EnTrack identifies the causes of energy consumption in system services that had previously remained obscure. We also introduce usage cases of EnTrack, which use it as a performance analysis tool. By discussing the cases, we demonstrate that EnTrack assists developers in improving the energy efficiency of applications and systems, and enables the investigation of possible energy leaks in Android services. We further show an EnTrack-based energy analysis, compared to similar tools [26-28], provides detailed information about the energy behavior of both applications and system services.

The contributions of our work are as follows:

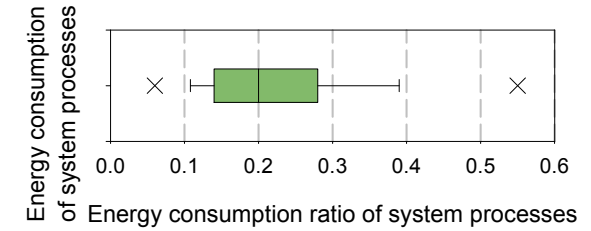
- We propose EnTrack, which is a facility to finely trace energy consumption of the system services requested by applications. To the best of our knowledge, this is the first work to understand the energy behavior of internal systems, i.e., system services, for the Android framework in particular.
- Based on our understanding of the energy behavior of system services, we specifically propose a scheme to account for the energy consumption of applications by considering their interactions with related system services, which have previously been concealed [14-17].

RELATED WORK

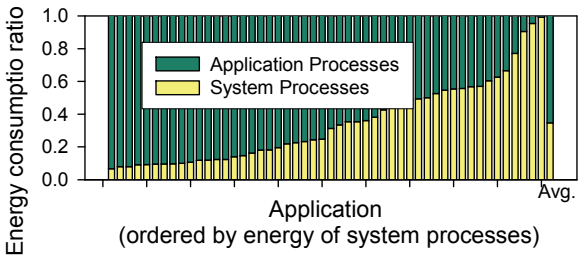
Energy accounting: Active work has been conducted to understand the energy consumption behavior of both applications and systems. PowerTutor [15] is an energy accounting tool that estimates energy consumption based on the power model of each hardware component. The method utilizes information obtained from *procfs/sysfs* in the Linux system and the *BatteryStat* information in the Android platform. Energy consumption is estimated for each application identified by its UID. The eProf [16] approach uses information acquired in system calls to estimate the energy consumption of applications. The scheme makes use of an FSM-based power model, which improves energy estimation by using the callee information detected in system calls. Dong et al. [14] uses a cooperative game theory for the energy accounting method. The scheme maintains high accuracy without requiring analysis on hardware characteristics. AppScope [17] tracks the kernel functions related to energy consumption of hardware components, and estimates application's energy consumption aggregated over UID.

Although energy accounting has been made increasingly accurate through active work in the field, one implicit assumption is that the total amount of energy consumed by an application is estimated by cumulating the energy consumed by the application processes and threads. In the case of an Android platform, this can be accomplished by identifying the application processes and threads via the UID. The assumption is, however, questionable because applications running on mobile systems typically rely on many system processes that perform specific services on their behalf. Compared to our previous work of AppScope, EnTrack estimates the energy consumption of system services invoked by applications, and enables a fine-grained monitoring in terms of function units, beyond the processes and threads (i.e., UID-based scheme).

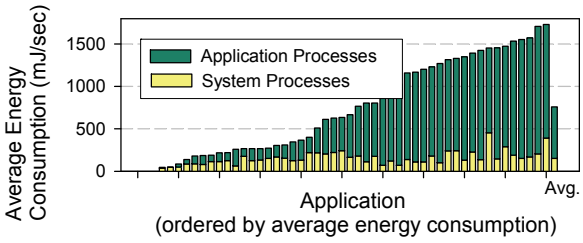
Control-flow tracing: A variety of research has been conducted to understand the complicated behaviors of distributed systems and to deal with failures and performance issues. For distributed systems, Chen et al. [18] introduced a runtime approach that automatically detects failures and manages performance issues that are possibly caused by a lack of knowledge on the system in the update process. Magpie [19] is a tool chain that traces the control flow of application requests and generates workload models of system behavior for performance prediction as well as anomaly detection. In the work, tracing was accomplished by analyzing the relations among events in each hardware



(a) Energy consumption of system processes on 17 devices



(b) Energy consumption ratio of 50 applications



(c) Energy consumption of 50 applications

Figure 2: Energy profile of system processes.

component based on a pre-defined event schema instead of propagating randomly assigned request identifiers. X-Trace [20] is a tracing framework for distributed systems that diagnoses performance problems. The system achieves this goal by tracking the data flow of data messages within the system. AppInsight [21] instruments mobile application binaries to evaluate application performance. The system identifies the critical paths in user transactions rather than the overall system. The scheme inserts additional code into the application instead of making changes to the system in its entirety. Power containers [11] traces the energy usage of individual requests in multi-core servers and aggregates the data to estimate the total energy consumption.

Previous works have mostly aimed to understand the internal behavior of systems and subsequently optimize system performance. Similarly, this paper focuses on understanding system behavior so as to optimize energy consumption, which is critical for mobile devices. Specifically, we trace the operation of system behaviors related to running applications and determine the causes of energy consumption in mobile platforms.

MOTIVATION

Modern mobile platforms, including Android, have a layer of middleware between the applications and the kernel. The layer contains various API functions and independently runs system processes, which are broadly referred to as *system services*. The purpose of the system services is to help simplify application development and achieve consistency and efficiency in managing physical resources. Thus, recent mobile platforms are considered forms of server–client systems or distributed systems. System processes in mobile platforms are in charge of managing physical resources as well as access requests from applications. The energy consumption of system processes should therefore be traced via the caller (i.e., application) to understand the entire energy consumption of mobile devices. However, the set of system services in commercial mobile platforms is a black box from which we cannot trace the detailed energy consumption of system services. Previous UID-based approaches [14-17] were not able to handle this black box since various system services run in one process. For example, the *system_server* process of the recent Android contains about 40 services, such as *ActivityManager*, *LocationManager*, and so on.

We therefore investigated the energy consumption of system services while an application was running. First, we collected energy consumption information based on the Battery menu in the Settings on Android phones from 17 users. The data were obtained with a power model-based estimation method similar to that used in previous works [15, 17]. Among the list of items in the Battery menu, “Android OS” and “Media Server” are the ones that spend energy on behalf of system processes. As shown in Figure 2(a), we found that a considerable portion of total energy (i.e., 22.2% (standard deviation (SD)=11.3)) were dedicated to these two items (i.e., system processes). However, it was unclear which applications consumed this amount of energy, or if indeed it was consumed by Android itself.

In addition, we analyzed the proportion of energy consumption attributable to the system processes. We observed energy consumption while running a set of 50 arbitrarily selected applications on a Nexus 4 device with Android 4.3 (Jelly Bean). For the analysis, we used the kernel monitoring-based energy accounting scheme proposed in [17]. We also measured the energy consumption of the CPU specifically because the CPU is the hardware component that can explicitly separate the attribution of energy consumption to both application processes and system processes. Figure 2(b) shows that on average 34.6% (SD = 24.7) of the energy consumed by a CPU is attributable to system processes. This proportion is relative to the total amount of energy consumed by the CPU. When compared with the actual values (i.e., the values with the Joule unit), 20.0% of the energy (SD = 10.9) on average was found to be attributable to the system processes, as shown in Figure 2(c). This result implied that a significant portion of energy consumption by system processes has not been accounted for in previous research.

Since a considerable portion of energy consumption is caused by system processes that support applications, it is

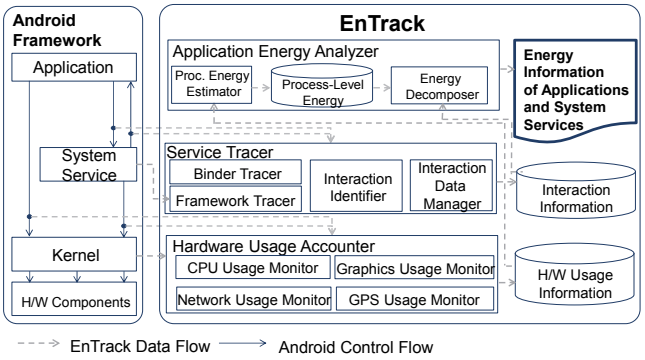


Figure 3: EnTrack architecture and approach overview.

reasonable to categorize the energy consumption as being the result of application processes; hence the energy behavior of system processes should be examined closely.

Our work next attempted to address two issues: (1) how to estimate the energy usage of system services by function units online, and (2) how to divide and attribute portions of energy consumption by system services to applications. These were challenging since system services are shared by many applications and various services run simultaneously.

ENTRACK

In this section, we describe the design and implementation details of EnTrack. EnTrack is an Android framework-based facility for tracing the control flow between applications and system services to provide the fine-grained energy consumption of both applications and system services. Through procedure-level tracing, EnTrack precisely analyzes the relationship between applications and system services in terms of energy consumption.

Figure 3 illustrates the overall architecture of EnTrack. EnTrack consists of three key components: Service Tracer, Process Usage Accounter, and Application Energy Analyzer. Service Tracer keeps track of applications' access to system services, and Process Usage Accounter logs the access to the physical resource by each process. Application Energy Analyzer combines raw data to produce meaningful information. These components of EnTrack work together to estimate the energy consumption of each application.

The Android platform offers numerous system services, the number of which varies depending on the hardware device or the OS version. In the case of Android 4.3 on Nexus 4, with which we implemented EnTrack, 74 system services were provided, and it was simply not practical to monitor every procedure in each of the 74 services. Fortunately, the system services in Android interact with applications through a common facility (i.e., Binder), thus EnTrack only had to monitor one point, located at the Binder Driver, to identify the interactions between the applications and the system services. Our implementation of EnTrack required only a few modifications to the system (i.e., we only added about 200 lines of code in the Android framework), and this had the added benefit of enhanced portability.

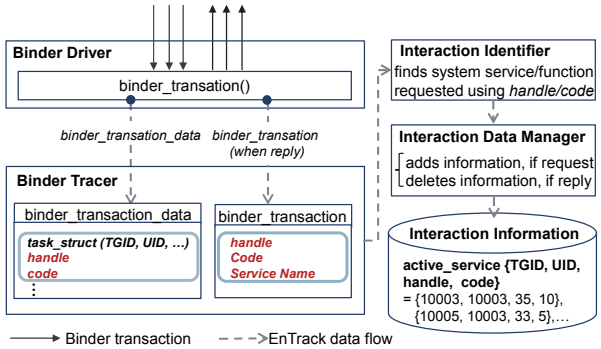


Figure 4: Service Tracer.

Service Tracer

Service Tracer collects the information necessary to monitor the caller of each system service and the service usage. The challenge was to enable fine-grained tracing with minimal overhead. The application uses various services within a few ticks (i.e., less than a millisecond), and diverse services run simultaneously. We propose a lightweight method that is finely-tuned in the Android framework to minimize the monitoring overhead.

Figure 4 shows the overall process of the component. Service Tracer specifically tracks the interactions exchanged between the applications and system services to collect the required data for EnTrack. This was accomplished by probing a set of specific internal operations of Binder IPC messages and system services. A system service for application starts when a request is received at the system service and a reply is sent back to the application through IPC messages in Binder after the task is finished. Our message hooking method in Binder is robust, lightweight, and runs online. Most of all, the method is highly portable in comparison with the component-level code modification approach generally used for control-flow tracing [11, 18-20]. Our approach does not require additional code modification for new components (i.e., new system services or functions) when porting to other devices or other versions of Android.

With the data collected via Binder Tracer, the following items were identified: the system service being used; the functions called in the system service; and the caller of the system service. Interaction Data Manager packages this information as Interaction Information. However, three kinds of necessary data (see the 'Framework Tracer' section) are not obtainable by only monitoring the messages passed through Binder. For example, in the case of the *LocationManager* service, the location acquisition API is highly abstracted and only a minimal amount of data is passed through Binder in messages. Detailed decisions, such as choosing a location medium between GPS or Wi-Fi, or when to register or deregister location listeners, are made inside *LocationManager*. In the case of graphics services, the operations are complicated because the speed of graphic processing varies and communication with hardware is conducted by interrupts. Therefore, examining the Binder

messages does not always generate all the required information. To address this issue, we implemented a separate system service called Framework Tracer to monitor the internal operations of the Android framework and append the necessary data to the Binder messages.

Binder Tracer

Binder Tracer monitors Binder Driver in the kernel, acquires the IPC messages exchanged between the applications and system services, and extracts relevant data to transfer to Interaction Identifier. We used jprobe [22] to hook into the function *binder_transaction()* in Binder Driver. By distinguishing the type of interactions and recording the timestamps at which each interaction takes place, it is possible to find out for how long a given application has used the specified system service.

EnTrack also requires information regarding the functions that are used in a given system service. To acquire this information, *binder_transaction_data*, the TGID, and UID of the involved application process are extracted, and this is then transferred to Interaction Identifier. The structure *binder_transaction_data* indicates which function was used with which arguments in which system service in the form of an index. After analyzing the received data, Interaction Identifier discovers the necessary information on the system service that received requests from the applications.

Interaction Identifier

To identify which application requested which system service and which function was called, Interaction Identifier analyzes the data transferred from Binder Tracer, including the structure *binder_transaction_data*, TGID, and UID of the involved application process.

First, when a request for a system service is made, the PID and UID of the process in the Binder message, which EnTrack monitors, are taken as the identifiers of the application. The information within the structure *binder_transaction_data* is analyzed to find out which system service is being requested. To use a system service, an application sends a request to *Context Manager* and obtains the handle and code of the system service, both of which can be used to identify the system service itself and the function in the service. When Binder Driver receives a message afterward, the discovered data (i.e., handle and code) is included in *binder_transaction_data* so that the request is relayed to the identified system service. Hence, we used the handle and code values to identify the system service and function being requested.

When a reply is made, the TGID and UID of the process that transmits the reply are used to identify the system service. The issue is complicated because there are service container processes, such as the system server or media server, that include a variety of system services within a single process. Hence, the information of the PID, TGID, or UID does not accurately reflect which system service is transmitting a reply. To derive the values of the handle and code in reply, we made a modification to the structure of *binder_transaction*. As a result, the handle, service name, and code of the service are stored when a request is made, and this makes it possible to determine which system service replied.

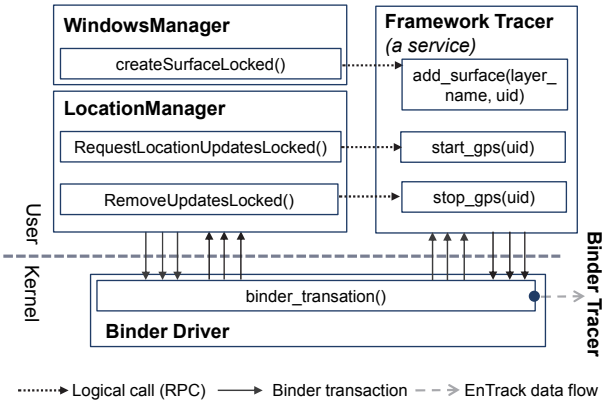


Figure 5: Framework Tracer.

Interaction Data Manager

Interaction Data Manager constructs Interaction Information to package the duration, handle, code of the system service, and pertinent data of the system service caller. Starting from the point at which a request is made, the TGID, handle, service name, and code become available, all of which are added to the active system service list. The PID and UID of the system service caller are recorded to the caller list of the system service. When a reply is made, the system service is searched for in the active system service list, and the application is deleted in the caller list. The additional data extracted from Binder messages by Framework Tracer is recorded by Interaction Data Manager.

Framework Tracer

Framework Tracer is a system service we implemented to obtain information that is not discoverable with the Binder messages. The overall architecture of Framework Tracer is shown in Figure 5. Within Framework Tracer, functions are defined that take arguments containing the information necessary to trace the energy of the system service. Within the services pertinent to the location and graphics processing, we added code that makes calls to function in Framework Tracer to allow data collection during Binder RPCs. *LocationManager* has two functions that are responsible for registering and deregistering location listeners, namely, *RequestLocationUpdatesLocked()* and *RemoveUpdatesLocked()*. We included calls to *start_gps()* and *stop_gps()* of Framework Tracer in both functions and passed as arguments the UID process/thread that owns the location listener. These calling activities generate Binder messages for Framework Tracer and detect the time points at which a location listener is registered/deregistered in order to notify Interaction Identifier. EnTrack attributes the energy consumption of the graphics component to applications according to the proportion of projected area on screen. To accomplish this, it is necessary to examine

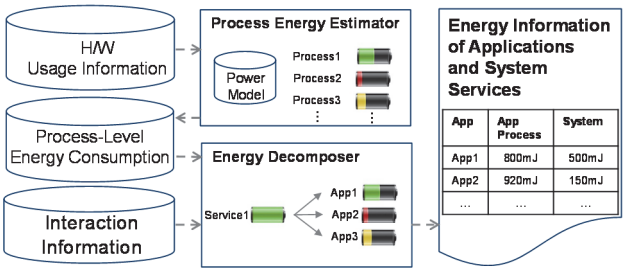


Figure 6: Application Energy Analyzer.

each layer that constitutes the screen and the owner of each layer. In Android, the *Hardware Composer* manages what is projected on screen for each region and layer through the object called Layer. To the code where Layer is constructed, i.e., at *WindowStateAnamator::createSurfaceLocked()*, we added a function call to *add_surface()* of Framework Tracer with the arguments of the layer name and UID. This enables Binder Tracer to keep track of the layer generation.

Hardware Usage Accounter

Hardware Usage Accounter collects usage data to determine how much energy is consumed by each hardware component. Many solutions have been proposed for this [15-17]. In the current implementation of EnTrack, the kernel and hardware drivers are monitored, in the same way as AppScope, to collect the usage data of the CPU, GPU, screen, and network.

In AppScope, hardware usage information is gathered for each process. The usage of graphics components or GPS is assigned *a priori* according to the assumption that the energy consumed by graphics components (i.e., GPU and screen) is attributed to the foreground application. What actually happens in the system, however, deviates from such assumptions. In Android, two or more applications could be responsible for what is projected on the screen concurrently, and location data could be shared by multiple applications. We modified AppScope to collect the usage data of graphics components and GPS. The data is then translated to energy units by Application Energy Analyzer and disaggregated to the applications.

Application Energy Analyzer

Based on Interaction Information as well as the hardware usage data, Application Energy Analyzer estimates the energy consumption by each application, including the energy accounted for in the system services. Figure 6 illustrates the overall scheme. First, Process Energy Estimator computes the energy consumption at the CPU and network, for each application process and system process, by using the power model used in AppScope. As explained above, the energy consumed for the location services and graphics-related services is estimated for the entire device. Based on Interaction Information, Energy Decomposer takes the sum of the energy consumption estimated for each hardware component (i.e., GPS and graphics) and disaggregates the consumed energy.

The following service containers are primarily driven by applications: *System Server*, *Media Server*, and *Surface Flinger*. Thus, the CPU usage by the processes themselves and the thread groups forked from them should be attributed to service caller applications. The energy usage data from the service containers are collected and disaggregated to applications.

We disaggregated the energy consumed for the graphics components based on the proportion of area taken by the application. To identify the proportion of area, EnTrack acquires data in the Layer object, which include *displayFrame*, z-order, and the Layer name. *displayFrame* contains the coordination of the application area, and z-order is the vertical ordering of the applications. By using the Layer list in Interaction Information, the name of each layer and UID are matched to identify the actual owner of each layer. With the layer region and z-order, the displayed area of each application is calculated, and the energy consumption at the GPU and the screen is disaggregated and attributed to the applications. In the implementation, the energy is attributed to the layer with the highest z-order if more than two layers overlap in a region. The Nexus 4 smartphone has an LCD display. Note, we do not consider an OLED display in the current work. Regarding GPS, we dissaggregate its energy consumption equally to applications which use GPS concurrently.

EVALUATION

In this section, we evaluate EnTrack in terms of overhead and functionality. We compared EnTrack with a UID-based scheme that considers the sum of the energy consumed by the application processes only [17]. We implemented EnTrack on the Nexus 4 device with Android 4.3.

Accuracy and Functionality

We validated the functionality of EnTrack according to each component: Service Tracer, Hardware Usage Accounter, and Application Energy Analyzer. We evaluated the performance of EnTrack in terms of accuracy and overhead, and presented the usefulness of EnTrack compared with the UID-based scheme. Note that a direct evaluation of energy accounting per application is not feasible. Hence, we chose an indirect evaluation method used in prior works [11, 14, 17, 23].

We first validated the performance of Service Tracer. The accuracy of Service Tracer is determined by whether EnTrack correctly monitors the system services requested by the applications. For this, we selected the top four system services that consumed the most energy in the first trial of the scenario. We added code to the services in order to collect logs on service usage in the scenario. That is, we manually collected service usage through source modifications. Then, we ran the scenario again and compared the service usage monitored by our scheme with that of the manual logging. Figure 7 shows the monitoring results for the four system services during the experiment. The results indicate that the monitoring result of EnTrack

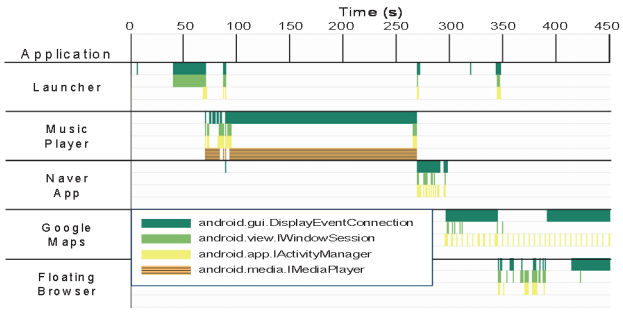
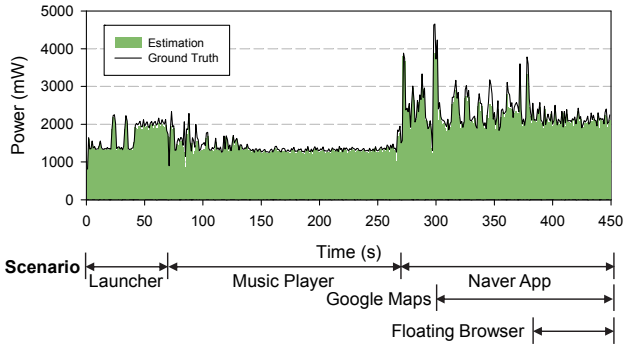
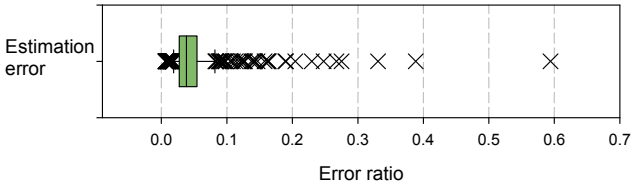


Figure 7: Trace of system services request by applications.



(a) Energy trace of the whole scenario



(b) Estimation error

Figure 8: Accuracy of estimation method

was exactly the same as the log. The trace is intuitively related to the events of the scenario that we performed. For example, the graphics system service *android.gui.DisplayEventConnection* was used for the frame buffer operations, and it constantly received requests while an application was projected on screen. Requests to *android.gui.IActivityManager* were observed when the current activity was switched, for example, by launching an application. Similarly, the requests to *android.media.IMediaPlayer* were continuously observed when Music Player was active.

We observed that the applied energy estimation method [17] in Hardware Usage Accounter produced highly accurate results. With Monkey [24], we executed five applications both sequentially and simultaneously according to a predefined usage scenario. Figure 8 shows the accuracy of our estimation method in the usage scenario. The ground truth values were measured with Monsoon Power Monitor [25]. Our implementation showed a margin of error of 4.94%, or 95.85mW, on average.

We evaluated the accuracy of Application Energy Analyzer. First, we conducted experiments to validate if the scheme

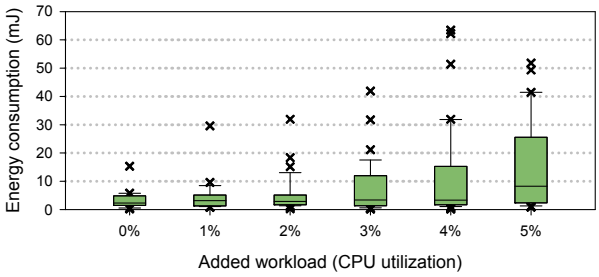


Figure 9. Energy distribution of each *getContentProvider()* function call.

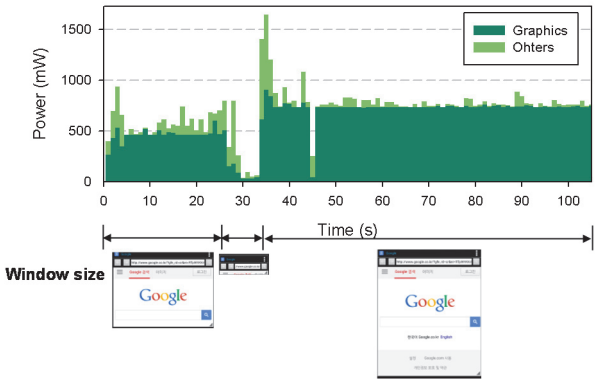
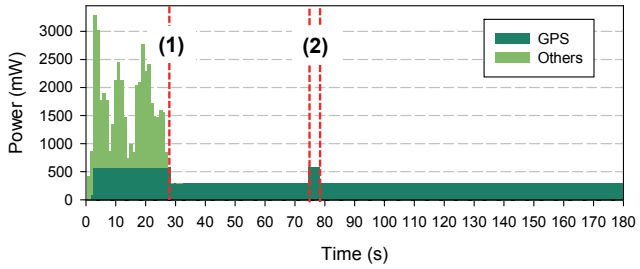


Figure 10: Disaggregation of energy consumed by graphics components while running Floating Browser.

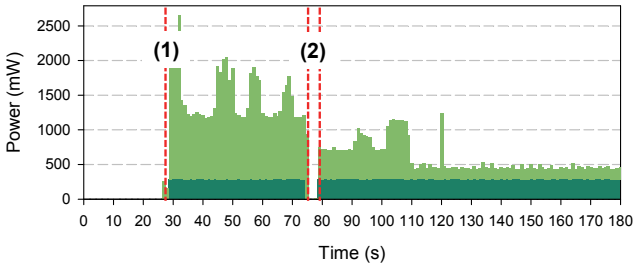
adequately disaggregates the CPU energy consumption of system services. We varied the CPU utilization at *getContentProvider()* in *android.gui.IActivityManager* which is called periodically by Google Maps. We then repeatedly ran Google Maps. Figure 9 shows the estimated energy consumption of the *getContentProvider()* function invoked by Google Maps. As the workloads increase, the estimated energy consumption also increases. This result validates that the proposed system accurately disaggregates the CPU energy consumption at function level.

We further validated the accuracy of Application Energy Analyzer by using two intuitive usage scenarios: that is, a graphics-related application and a location-based service application. Figure 10 shows the energy consumption by the graphics components while running Floating Browser with which the user resizes the displayed area on the screen. We varied the screen size for 45 seconds upon launching the application. As shown in the figure, the energy consumption level fluctuated during this period. Such consumption could not be observed in previous research since the entire energy consumed for graphics was attributed to the foreground application only.

Similarly, Figure 11 illustrates the energy consumption disaggregated and attributed to location service. Figures 11(a) and (b) show energy consumption while running Naver App and Google Maps, respectively. Both applications constantly requested the location service, and they ran simultaneously during some periods. Naver App was launched first, and 27 seconds later, Google Maps was



(a) Naver App



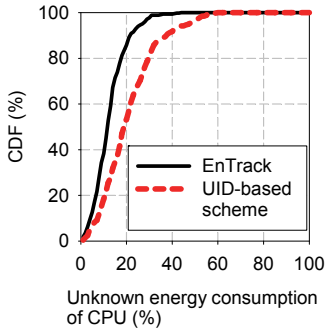
(b) Google Maps

Figure 11: Disaggregation of energy consumed by location services. At point (1), Google Maps was launched. At point (2), Home button was pushed and Google Maps was re-launched.

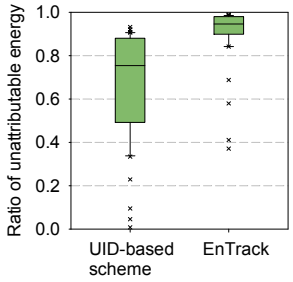
launched too, after which the location data were shared by both for 49 seconds. The charts show that the energy consumed by GPS was evenly disaggregated and attributed to both applications during the overlap, whereas this energy was attributed solely to Naver App when Google Maps stopped running briefly.

We next validated the usefulness of EnTrack, in comparison with an UID-based scheme (e.g. AppScope). Figure 12(a) shows the portions of CPU energy consumption not attributable to any application, both in the UID-based scheme and EnTrack. The x-axis of the chart indicates the relative proportion of the CPU energy consumption not attributed to any application. In 80% of the cases in the chart, approximately 30% of the energy consumption is unaccounted for in the UID-based scheme, whereas EnTrack left only 18% unexplained. This result indicates that EnTrack is superior to the existing UID-based scheme in terms of its capability of explaining CPU energy usage.

For a more general validation of this aspect, we tested EnTrack with the 50 application scenarios that were used in the preliminary experiment. Figure 12(b) shows the analysis capability of EnTrack when running the applications. The analysis capability is measured with the ratio of disaggregatable and attributable amounts of energy to the total amount of energy consumed by the CPU. Across the 50 different scenarios, the capability was improved to 90.8% (SD = 13.0) when using EnTrack, compared to 65.4% (SD = 24.7) with the UID-based scheme, and the performance variation between the usage scenarios was significantly reduced. The result indicates that EnTrack significantly



(a) Proportion of CPU energy consumption not attributable to any application (EnTrack vs. UID-based scheme (e.g. AppScope))



(b) Distribution of the unattributable energy proportions of 50 applications

Figure 12: Comparison of the analysis capability of energy consumption using EnTrack and the UID-based scheme.

improves energy accounting compared to the existing method. Note that the system processes used by the kernel were not considered in our study.

In summary, we observed that EnTrack (1) correctly traces the system services requests by applications, and (2) appropriately disaggregates and attributes energy consumed by the system services to each application.

Overhead

To measure the overhead of EnTrack, we compared the CPU utilization with and without running EnTrack during the usage scenario. We collected data using the K-best measurement scheme [30] with K = 3, ε = 0.05, and M = 20 with the repeated execution of the scenario. Our measurement showed that the EnTrack overhead is approximately 1.8% of CPU utilization. This result indicates that the overhead of EnTrack is negligible in terms of overall CPU utilization.

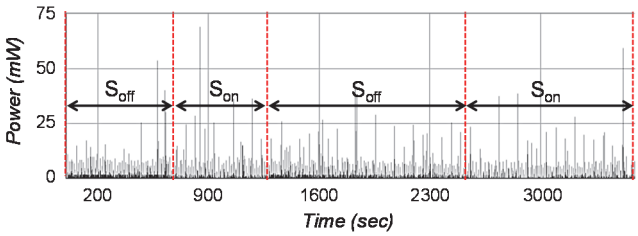
Meanwhile, the memory overhead of the current version of EnTrack was rather high. The log size for the 450-second scenario was 581KB. This overhead might be insignificant when EnTrack information is used for a developer's tools, such as the one introduced in next section; however, considering other potential applications, such as an operating system facility for OS-level power management, it should be further optimized.

CASE STUDY: DEVELOPER TOOL FOR ENERGY OPTIMIZATION

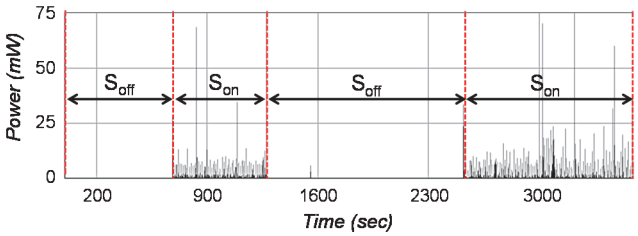
EnTrack can be used for energy optimization and fault detection, which commonly happens in applications and systems [18, 20]. In this section, we present the cases of real applications.

Case 1: Energy Optimization of LBS Application

The application for the case study is a location-based service [32] distributed on the market. This application continuously monitors the current user location by using the background GPS. The application was developed using



(a) Old version



(b) New version

Figure 13: Comparisons of energy consumption of *android.app.IActivityManager* for our application (S_{on}: Screen on, S_{off}: Screen off).

AlarmManager in Android to periodically turn on sensors (e.g., every 10 minutes) to estimate a user's location. With EnTrack, we found that the application consumed a considerable amount of energy with the *android.app.IActivityManager* system service. For a detailed analysis, the power trace in Figure 13(a) shows that a significant amount of energy is consumed by the service. We found out that the function *finish_receiver()* consumes most energy, which is used in broadcast receiver and *AlarmManager*. In the updated version, we found that the application now activates sensors with *Handler* instead of *AlarmManager*, which uses a broadcast receiver. EnTrack shows that the energy consumption of the updated version has successfully been reduced. That is, the system service *android.app.IActivity* consumed significantly less energy than the old version, as shown in 13(b). The disadvantage of *Handler* is that the operation is delayed when the phone screen is turned off, but such a delay is trivial since the operations could be performed when a user is using the phone. In summary, we were able to understand the energy efficiency of this application by using EnTrack.

Case 2: Energy Optimization of Web Browser

We further analyzed the energy consumption of the system service which is used in the Firefox application. The application provides tab switching for easy access of multiple web pages. Since the tab switching is commonly used in web browsing, the developers of Firefox improved its performance by updating the application at version 26.0. We profiled version 26.0 as well as previous version to observe the change of energy usage related to tab switching.

Figure 14 shows the energy consumption of tab switching in version 26.0 and 25.0 of the Firefox application. The overall energy usage decreased from 90,000 mJ to 65,800 mJ. In version 26.0, Firefox does not consume any energy

	Service Name	Energy (mJ)	Ratio (%)
1	android.gui.DisplayEventConnection	4016.88	4.46
2	android.view.IWindowSession	2844.15	3.16

(a) Version 25.0

	Service Name	Energy (mJ)	Ratio (%)
1	android.view.IWindowSession	2797.18	4.25
2	android.media.IAudioService	1441.13	2.19

(b) Version 26.0

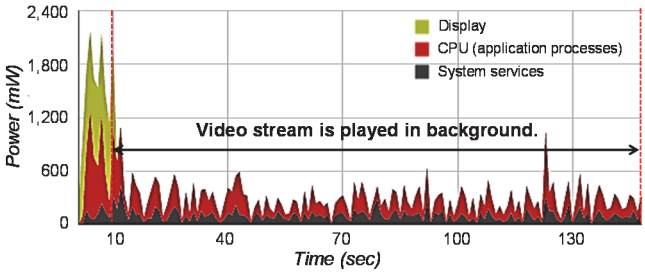
Figure 14: Comparisons of energy consumption of system services for Firefox application.

Service	Energy (mJ)
android.hardware.IOMX	10131.66
android.media.IAudioPolicyService	4719.09

(a) The energy consumption of system services in Media Server

Application	Energy (mJ)
org.mozilla.firefox.sharedID	10131.66

(b) Applications that trigger energy consumption in *android.hardware.IOMX*



(c) Energy trace of Firefox

Figure 15: Analysis of the Firefox bug.

at android.gui.DisplayEventConnection, which consumed the most energy at version 25.0. The reason is that the bug is fixed at version 26.0, *"Improved page load times due to no longer decoding images that aren't visible"* [29]. In detail, when user switches tabs, version 25.0 loaded the images in all web pages although some of pages are not visible to the user. However, in version 26.0, Firefox only loaded the images visible to the user. This significantly reduced the workload for loading images by surfaceFlinger. In summary, EnTrack enables the understanding of such internal behavior in system services, in terms of energy usage.

Case 3: Analyzing Energy Bug of Web Browser

A recent study [31] reported an energy bug that the Firefox application continues to operate the timer or video on a webpage even when the application runs in the background. We now present that EnTrack enables a detailed analysis of energy consumption in the bug-fixing process.

Figure 15(a) shows an analysis of the Firefox bug with EnTrack. EnTrack detected a non-trivial amount of energy consumption (i.e., 10.1J) with the *android.hardware.IOMX* system service in the Media Server process (also a service container in this case). Figure 15(b) shows that it is possible for the developer to ascertain that *android.hardware.IOMX* is being used by Firefox. In the energy trace (Figure 15(c)), we see that *android.hardware.IOMX* continuously consumes a certain amount of energy although the application does not consume energy for display. In other words, Firefox still uses the system services while the application is running in the background.

Existing studies [14-17] and the battery information provided by the Android framework consider only the energy consumption of the application processes; hence, the method cannot exactly identify the problematic situations derived by different services usage. In contrast, EnTrack systematically traces energy efficiency and *investigates the cause of the energy bugs*.

DISCUSSION

In this section, based on our findings, we further discuss the usability of EnTrack in terms of the energy optimization of mobile devices. We then describe the limitations of our work and some suggestions for future research.

Energy Optimization of System Services

EnTrack certainly improves our understanding of the behavior of system services in relation to energy. Until now, the services in middleware (e.g., Android) were almost a black box for application developers [3-6]. Our experiments with EnTrack showed that the portion of energy consumed by system services is 34.6% on average and 90% at most. The result indicates that energy optimization should be considered in both applications and system services. We advocate that the energy efficiency of system services is in fact more critical than that of the applications themselves since the system services are widely used by applications. We believe that EnTrack could be used as a powerful tool for both system developers and application developers.

Guidance of APIs in System Services by Energy Consumption

In our case study, we found that the energy consumption of an application could be reduced by about 46.8% by changing the function of the system services (i.e., from *AlarmManager* to *Handler* in our example). This result implies that EnTrack could be used to provide energy awareness in the use of APIs in system services. Mobile platforms document API usage, but the energy consumption of each function is usually not detailed. Developers have several ways to implement specific procedures. If the document on API usage were to contain the energy consumption of each function, it would be truly useful when designing energy-efficient applications.

Limitations and Future Work

In the present work we analyzed the Android system services and showed that the analysis capability using our method is superior to conventional methods. However, system processes, such as the ones used by the kernel, were not considered in our study despite the fact that they contribute to overall energy consumption. For example, our experiment with the 50 applications showed that about 9.2% of the energy consumption of an entire system is still concealed with the worst case of 62.9%. Also, EnTrack is currently implemented on the Android platform. Although our technique is generic enough to be applied to other systems such as Windows [33], Tizen [34], and iOS [35], further effort is needed to validate the scheme on various platforms.

The accuracy of an energy accounting scheme is usually validated via an indirect method [11, 14-17, 23]. That is, many studies [11, 17, 23] only compared the summation of the energy consumption of software units with the measurement results of an entire system for the validation of accuracy. This is because there is no known technique to find the ground truth of the energy consumption of applications or processes [14]. Dong et al. [14] claimed that a theoretical ground truth for energy accounting could be obtained by a cooperative game theory. Although their work provides insight into the per-software-unit energy accounting scheme, it is probably not feasible to acquire the theoretical ground truth for more recent, complicated mobile systems, which potentially have hundreds of processes running at the same time. A more pragmatic method should therefore be devised to measure the energy consumption of each software unit.

CONCLUSION

In this paper, we proposed EnTrack, an energy tracing facility for the Android framework. EnTrack keeps track of interactions between applications and system services by monitoring and analyzing the messages of Binder, which is the RPC mechanism of the Android framework. The system then estimates the energy consumption of each system service and disaggregates the energy consumed by the system services to each application based on intuitive guidelines. Through an extensive experiment, we showed an improvement in the analysis capability of energy consumption using EnTrack over the existing method. We also introduced a usage case of EnTrack as a developer tool for energy optimization, and demonstrated that it is possible to solve practical energy issues in both applications and system services, such as optimizing the energy consumption in applications, analyzing energy bugs in applications, and validating the energy consumption of system services.

Ultimately, this study is an effort to understand system energy consumption more thoroughly and make energy accounting for applications more accurate. However, the current implementation of EnTrack has a few limitations, which can be remedied. In our future work, we plan to focus on system processes other than system services and develop an enhanced tracing method that takes into account their contribution to energy consumption.

ACKNOWLEGEMENT

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 (NRF-2014R1A2A1A11049979).

REFERENCES

1. Rahmati, A., Qian, A., and Zhong, L. Understanding human-battery interaction on mobile phones. In *Proc. MobileHCI 2007*, ACM Press (2007), 265-272.
2. Truong, K. N., Kientz, J. A., Sohn, T., Rosenzweig, A., Fonville, A., and Smith, T. The design and evaluation of a task-centered battery interface. In *Proc. UbiComp 2010*, ACM Press (2010), 341-350.
3. Ma, X., Huang, P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L. K., and Voelker, M. eDoctor: automatically diagnosing abnormal battery drain issues on smartphones. In *Proc. NSDI 2013*, USENIX (2013), 57-70.
4. Oliner, A. J., Iyer, A. P., Stoica, I., Lagerspetz, E., and Tarkoma, S. Carat: collaborative energy diagnosis for mobile devices. In *Proc. SenSys 2013*, ACM Press (2013).
5. Ferreira, D., Ferreira, E., Goncalves, J., Kostakos, V., and Dey, A. K. Revisiting human-battery interaction with an interactive battery interface. In *Proc. UbiComp 2013*, ACM Press (2013), 563-572.
6. Mittal, R., Kansal, A., and Chandra, R. Empowering developers to estimate app energy consumption. In *Proc. MobiCom 2012*, ACM Press (2012), 317-328.
7. Li, D., Hao, S., Halfond, W. G. J., and Govindan, R. Calculating source line level energy information for Android applications. In *Proc. ISSTA 2013*, ACM Press (2013), 78-89.
8. Lee, S., Yoon, C., and Cha, H. User interaction-based profiling system for Android application tuning. In *Proc. UbiComp 2014*, ACM Press (2014), 289-299.
9. Flinn, J., and Satyanarayanan, M. Energy-aware adaptation for mobile applications. In *Proc. SOSP 1999*, ACM Press (1999), 48-63.
10. Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. ECOSystem: managing energy as a first class operating system resource. In *Proc. ASPLOS X*, ACM Press (2002), 123-132.
11. Shen, K., Shriraman, A., Dwarkadas, S., Zhang, X., and Chen, Z. Power containers: an OS facility for fine-grained power and energy management on multicore servers. In *Proc. ASPLOS 2013*, ACM Press (2013), 65-76.
12. Martins, M., and Fonseca, R. Application modes: a narrow interface for end-user power management in mobile devices. In *Proc. HotMobile 2013*, ACM Press (2013).
13. Zhang, N., Ramanathan, P., Kim, K. H., and Banerjee, S. PowerVisor: a battery virtualization scheme for smartphones. In *Proc. MCS 2012*, ACM, New York, NY, 37-44.
14. Dong, M., Lan, T., and Zhong, L. Rethink energy accounting with cooperative game theory. In *Proc. MobiCom 2014*, ACM Press (2014), 531-542.
15. Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. CODES+ISSS 2010*, ACM Press (2010), 105-114.
16. Pathak, A., Hu, Y. C., Zhang, M., Bahl, P., and Wang, Y. Fine-grained power modeling for smartphones using system call tracing. In *Proc. EuroSys 2011*, ACM Press (2011), 153-168.
17. Yoon, C., Kim, D., Jung, W., Kang, C., and Cha, H. AppScope: application energy metering framework for Android smartphones using kernel activity monitoring. In *Proc. USENIX ATC 2012*, USENIX (2012), 387-400.
18. Chen, M. Y., Accardi, A., Kıcıman, E., Lloyd, J., Patterson, D., Fox, A., and Brewer, E. Path-based failure and evolution management. In *Proc. NSDI 2004*, USENIX (2004), 309-322.
19. Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for request extraction and workload modelling. In *Proc. OSDI 2004*, USENIX (2004), 259-272.
20. Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-Trace: a pervasive network tracing framework. In *Proc. NSDI 2007*, USENIX (2007), 271-284.
21. Ravindranath, L., Padhye, J., Agarwal, S., Mahajan, R., Obermiller, I., and Shayandeh, S. AppInsight: mobile app performance monitoring in the wild. In Proc. *OSDI 2012*, USENIX (2012), 107-120.
22. Keniston, J., Panchamukhi, P., and Hiramatu, M. Kernel Probes (Kprobes). Retrieved February 21, 2015 from http://www.kernel.org/doc/Documentation/kprobes.txt.
23. Fonseca, R., Dutta, P., Levis, P., and Stoica, I. Quanto: tracking energy in networked embedded systems. In *Proc. OSDI 2008*, USENIX (2008), 323-338.
24. UI/Application Exerciser Monkey. Retrieved February 21, 2015 from http://developer.android.com/tools/help/monkey.html.
25. Monsoon Solutions Inc. Power Monitor. Retrieved February 21, 2015 from https://www.msoon.com/LabEquipment/PowerMonitor.
26. Traceview. Retrieved February 21, 2015 from http://developer.android.com/tools/help/traceview.html.
27. Qualcomm Technologies, Inc. Trepn Profiler. Retrieved February 21, 2015 from

https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler.

28. ARM Ltd. Optimize: Streamline Performance Analyzer. Retrieved February 21, 2015 from http://ds.arm.com/ds-5/optimize/.
29. Bug 847223 - Don't decode images that aren't visible when we download them. Retrieved February 21, 2015 from https://bugzilla.mozilla.org/show_bug.cgi?id=847223.
30. Bryant, R. E. and O'Hallaron, D. R. *Computer Systems: A Programmer's Perspective*. Prentice Hall (2004).
31. Liu, Y., Xu, C., and Cheung, S. -C. Characterizing and detecting performance bugs for smartphone applications. In *Proc. ICSE 2014*, ACM Press (2014), 1013-1024.
32. LifeMap. Retrieved February 21, 2015 from https://play.google.com/store/apps/details?id=com.mobed.lifemap
33. Windows Phone. Retrieved June 21, 2015 from https://www.windowsphone.com/
34. Tizen. Retrieved June 21, 2015 from https://www.tizen.org/
35. iOS. Retrieved June 21, 2015 from https://www.apple.com/ios/