AppScope: Application Energy Metering Framework for Android Smartphones using Kernel Activity Monitoring

Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, Hojung Cha
Dept. of Computer Science,
Yonsei University, Korea
{cmyoon,dkim010,wwjung,ckkang,hjcha}@cs.yonsei.ac.kr

Abstract

Understanding the energy consumption of a smartphone application is a key area of interest for end users, as well as application and system software developers. Previous work has only been able to provide limited information concerning the energy consumption of individual applications because of limited access to underlying hardware and system software. The energy consumption of a smartphone application is, therefore, often estimated with low accuracy and granularity. In this paper, we propose AppScope, an Android-based energy metering system. This system monitors application's hardware usage at the kernel level and accurately estimates energy consumption. AppScope is implemented as a kernel module and uses an event-driven monitoring method that generates low overhead and provides high accuracy. The evaluation results indicate that AppScope accurately estimates the energy consumption of Android applications expending approximately 35mW and 2.1% in power consumption and CPU utilization overhead, respectively.

1 Introduction

With the widespread use of smartphone applications, the energy consumption of each application is important information that is used to manage a device's power. Smartphone users can adaptively select energy-efficient applications based on the energy consumption of an application. Additionally, understanding the energy consumption of each process or hardware component is a key area of interest for application and system software developers [1-5].

Estimating the energy consumption of a smartphone application is practically difficult. The estimation system should be able to determine the power usage of various hardware components in the device. However, this information is difficult to acquire because of complicated hardware schematics and compact form factor. An accurate power model for hardware components should be available to determine the level of energy consumption for each component.

Previous work has addressed various energy metering methods for smartphones [6-9]. However, these earlier models are lacking in terms of granularity and accuracy. An example of a previous system is PowerScope [6], which provided the energy consumption of applications at a fine-grained level, but required post-processing using an external device. The developer also needs to import related APIs for energy metering. PowerTutor [7, 8] proposed an estimation method for hardware components, but the system does not provide energy information for each application or process. PowerProf [9] required information on the API usage for each application in order to estimate energy consumption. The limitations of the above mentioned models are the result of schemes that focus on the accuracy of the power model but do not consider the actual usage of the hardware component. Accurate estimation of an application's energy consumption depends on the accuracy of the power model and the accuracy of a hardware component's usage statistics. In particular, the hardware usage estimation is critical because it is a pre-requisite for estimating the energy consumption of smartphone applications.

Usage estimation for hardware components has been previously completed using hardware performance counter (HPC) [5, 10-13], software performance counter (SPC) such as the Linux *procfs/sysfs* [7, 14-16], or *BatteryStats*, which is provided by Android [8, 17]. However, depending on the process or underlying hardware components, these approaches provide different information. Thus, obtaining accurate usage statistics for each hardware component is limited by this feature.

In this paper, we propose a software scheme, called AppScope. This scheme automatically estimates the energy consumption of applications running on Android smartphones. The proposed system accurately estimates the usage (or utilization) statistics for each device component. We have designed the scheme based on monitoring the Android kernel at a microscopic level. In order to estimate the usage statistics of each application, the system analyzes the traces of a system call, as well as the messages for Android binder inter-process communication (IPC). AppScope collects usage information based on an event-driven approach; hence, the energy consumption of each application is estimated at a fine-grained level. Additionally, the proposed approach is

applicable for any Android-based device, without modification of system software, because we implemented the scheme using a dynamic module in the Linux kernel.

The contributions of our work are as follows:

- AppScope provides the energy consumption of Android applications automatically, being customized to the underlying system software and the hardware components in the device.
- AppScope accurately estimates, in real-time, the usage of hardware components at a microscopic level.
- We implemented AppScope as a loadable kernel module to improve portability of the proposed approach. Thus, AppScope can be used on an Android-based device without modifying the system software.

2 Backgrounds

The accuracy of application energy metering and granularity of measurement depend on power and energy models. In this section, we discuss the models and briefly discuss DevScope, which provides a nonintrusive, online power analysis of smartphone hardware components.

2.1 Power and Energy Models

Depending on the power interdependencies among underlying hardware components, power models are typically classified into a linear or a non-linear regression model. The non-linear models often capture power dependency among hardware components, although their performance does not significantly outperform linear models [18]. We only consider linear models in this paper.

With a linear model, the power consumption P of a device is expressed as follows:

$$P = \Sigma_i(\beta_i \times x_i) + P_{base} + P_\epsilon \quad (1),$$

Here, x_i represents the vector of usage measurement for hardware component i and β_i the power coefficient for component i. Also, P_{base} is the base power consumption, and P_ϵ is a noise term that cannot be estimated by the model. Then, the total energy consumption E of a smartphone is expressed as:

$$E = \sum_j E^j + (P_{base} + P_\epsilon) \cdot D, \text{ where}$$

$$E^j = \sum_i(\beta_i \times x_i^j) \cdot d_i^j \quad (2)$$

Here, D is the device's power-up duration. E^j is the energy consumed by process j. E^j is expressed with β_i, x_i^j, and d_i^j, where x_i^j and d_i^j represent the usage vector and active duration of hardware component i accessed by process j, respectively. Note that the accuracy of E^j is influenced by β_i, x_i^j, and d_i^j. To estimate the energy consumption of smartphone applications, it is essential to obtain accurate values of β, x, and d in an effective way. Note that AppScope employs a linear model to estimate the energy consumption of smartphones.

2.2 DevScope

Previous studies on linear power modeling for mobile devices [7, 14, 15] used external power measurement to profile β_i for each device type. In practice, β_i varies, even on the same type of device, depending on hardware, software configuration, and battery status [16, 19]. Typically, these values are directly obtained with hardware measurements for target devices; hence, this offline method is costly and hardly adaptive to changing environments.

The limitation of the offline method can possibly be overcome by using an online approach that employs a battery monitoring unit (BMU) [16, 19], which is built in to smartphones. The scheme would enable the implementation of an online power model that automatically constructs a power model for each device, adapting to changes of external factors, such as aging or software updates. However, in order to employ a BMU as an online power measurement tool, we must consider two factors that are inherent in the properties of BMU. First, the information update rate of a BMU is noticeably lower than external measurement tools; hence the online results may not be accurate. Second, since the user is not able to intervene in the process of constructing a model, it is difficult to understand the exact relationship between system activities and power consumption.

DevScope [19], an Android application, is an automatic and online tool used to generate a power model for smartphones. The tool probes operating systems to obtain information about individual component types and their configurations. Additionally, by monitoring the update activity of BMU, DevScope detects the update rate automatically. According to individual component types, system configuration, and BMU update rates, DevScope dynamically creates a control scenario for each hardware component to perform power analysis. Hence, even though a device (i.e., smartphone) is identical, the scenario might be different due to each device's configuration. The scenario assigns a workload to each component, which then triggers every possible power state of the component; for example, specific operations for CPU, display brightness, GPS on/off, and packet transmission for cellular and WiFi. Each workload is

maintained for a time period to collect enough measurement samples (i.e., 5 samples) to overcome the limitation of BMU's low update rate. DevScope turns off every other component, except for the component under measurement. However, since the CPU should be alive to measure the power consumption of other component, CPU power analysis is conducted ahead of other hardware components. The power analysis of other hardware components is then conducted by subtracting the power consumption of the CPU from the total power consumption of the device that is being measured by the BMU. While performing the test scenario, DevScope classifies the results into each term of the power model and then generates corresponding power coefficients. DevScope requires a user's explicit interaction to initiate training and collect power coefficients. Hence, if re-training is required due to changes in a system's configuration, the process should be repeated manually, yet the power coefficients will be updated automatically. Note that the training time depends on the characteristics of underlying hardware components, as well as the update rate of BMU. The process typically takes minutes.

Currently, DevScope uses the device power model, illustrated in Table 1, for five core hardware components of smartphones, that is CPU, display, cellular (3G), WiFi, and GPS, and generates their power coefficients β_i. To analyze the CPU characteristics, DevScope locates the frequency-voltage table using */sysfs*; thus the number of available frequencies is dynamically determined. The power consumption is then measured by setting the frequency to every value.

In the case of display, DevScope presently only supports LCD displays, not more modern display types, such as OLED. The tool dynamically generates a table that contains coefficients for every possible brightness level. This is because the relationship between power consumption and brightness level is not completely linear [15].

To determine the coefficient for cellular, DevScope considers power consumption of each RRC (radio resource control) state; IDLE, FACH, and DCH (see Section 4.5 for further details). The power state transition is proven via a planned scenario in which data traffic is controlled. The power consumption pattern of WiFi differs depending on the specific packet rate (i.e., threshold workload size) [7]. DevScope gradually increases the packet rate and finds the threshold value at which the power consumption pattern is changed. DevScope currently uses a fixed-strength signal model for both WiFi and 3G; hence, although the model would suit the purpose of application and system developers, the tool should be supplemented to reflect true mobile environments.

The power states of GPS are defined into three states:

Table 1: Power model for smartphone components

Component	Model
CPU	$P^{CPU} = \beta_{freq}^{CPU} \times u + \beta_{freq}^{idle}$ u: utilization, $0 \le u \le 100$ $freq$: frequency index, $freq = 0,1,2 \cdots, n$
LCD	$P^{LCD} = \beta_b^{LCD}$ b: brightness level, $\mathrm{MIN}(level) \le b \le \mathrm{MAX}(level)$
WiFi	$P^{WIFI} = \begin{cases} \beta_l^{WIFI} \times p + \beta_l^{base}, & if\ p \le t \\ \beta_h^{WIFI} \times p + \beta_h^{base}, & if\ p > t \end{cases}$ p: packet rate, t: threshold
cellular(3G)	$P^{3G} = \begin{cases} \beta_{IDLE}^{3G}, & if\ RRC\ state\ is\ IDLE \\ \beta_{FACH}^{3G}, & if\ RRC\ \text{state}\ is\ FACH \\ \beta_{DCH}^{3G}, & if\ RRC\ state\ is\ DCH \end{cases}$
GPS	$P^{GPS} = \beta_{on}^{GPS},\ if\ GPS\ is\ on$

OFF, SLEEP, and ACTIVE. Since the switch between SLEEP and ACTIVE states has a constant pattern, we regard SLEEP and ACTIVE states as ON.

The goal of AppScope is to provide a practical application energy metering system that is readily runnable on Android smartphone. AppScope estimates energy consumption E^j of each process based on a linear model, as shown in Equation (2). AppScope employs DevScope's component power model (see Table 1) and the power coefficient $\beta_{i,}$ which are obtained for target devices. Therefore, in this paper we focus on the AppScope features that deal with the automatic acquisition of x_i^j and d_i^j for each hardware component accessed by an application.

3 AppScope: The Application Energy Metering System

AppScope is an application energy metering framework for the Android system that uses hardware power models and usage statistics for each hardware component. AppScope provides accurate and detailed information on the energy consumption of applications by monitoring kernel activities for hardware component requests.

Figure 1 shows an overview of AppScope. The system conducts application energy metering via three phases:

(1) Detection of process requests that are accessing hardware components.

(2) Analysis of usage statistics and status changes of the requested hardware components.

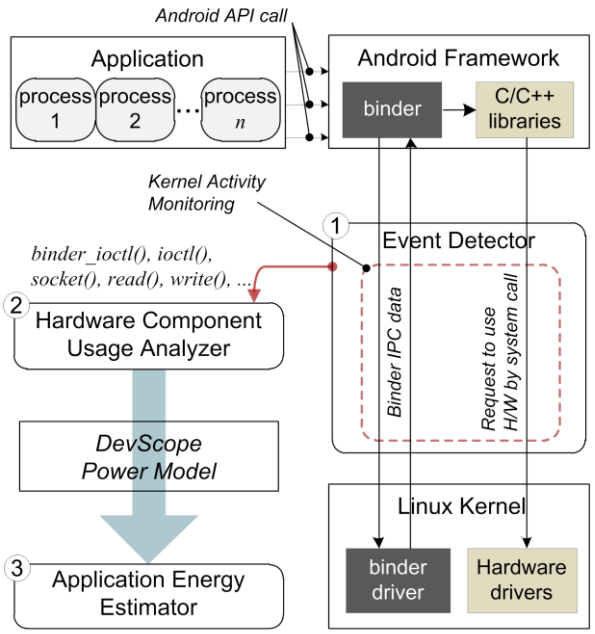


Figure 1: AppScope overview on Android platform.

(3) Linear model-based application energy estimation by adding up energy consumption of each hardware components accessed by application.

3.1 Event Detector

Event Detector probes system calls that are relevant to the hardware component operation, such as CPU frequency switching, process switching, packet transmission, and binder I/O control. Event Detector monitors *cpufreq_cpu_put()* for CPU frequency switching, and *sched_switch()* for process switching. For packet transmission operations, the usage of the *dev_queue_xmit()* and *netif_rx()* kernel functions are monitored. For binder I/O control, Event Detector monitors *binder_transaction()* which is a part of *binder_ioctl()* routine. These detections are passed onto the Hardware Component Usage Analyzer.

3.2 Hardware Component Usage Analyzer

When an event is detected, the Hardware Component Usage Analyzer collects usage statistics for each hardware component and data that is required to apply the power model to the component. Each hardware component is activated by different kernel operations. Moreover, the type of information required to apply the power model varies depending on the characteristics of power consumption. Therefore, the method of collecting information is separately defined according to the hardware components. Figure 2 illustrates different methods for the components. In the case of the CPU, the

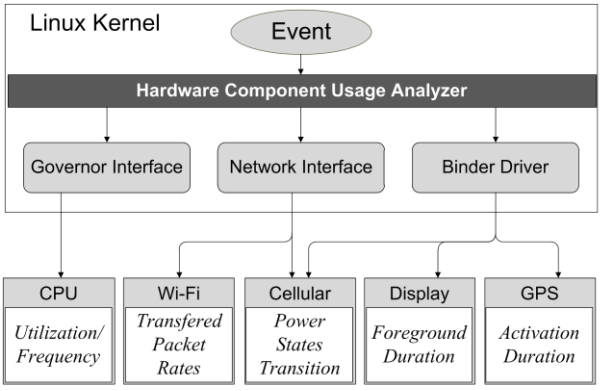


Figure 2: Hardware Component Usage Analyzer

changes in utilization and frequency are collected by referring to the governor interface. For WiFi, the rate of the transmitted/received packets of a process is collected by monitoring the data flow in the Linux networking stack. In the case of LCD display and GPS, the duration of activation is investigated by analyzing the IPC interfacing message of the Android binder. For 3G interface, the information on transmitted/received packets and the changes in power state are collected through the Linux networking interface and the Android IPC binder interface, respectively. The detailed process for each hardware component is presented in Section 4.

3.3 Application Energy Estimator

The energy consumption of an Android process is estimated via hardware usage statistics, which are applied to the underlying power model for hardware components (see Table 1). The application energy consumption is then obtained by combining the energy consumption of all processes that belong to an application. In the Android platform, each application has a unique user id (UID) to prevent other applications from accessing its specific resources. AppScope differentiates the energy consumption of an application using UID.

In our work, we assume that the overall energy consumption of a device running an application includes both "system energy" and "application energy". System energy is defined as a basic consumption that is required to operate a device using the Android framework. It includes the energy consumption for various Android system processes as well as for the Linux kernel threads. Meanwhile, application energy is defined as consumption solely used by the processes belonging to an application. In terms of UID in the current Android framework, UID=0 is used by the root-owned processes, the UIDs around 1,000 are used by the Android system processes, and the UIDs over 10,000 are used by applications. AppScope estimates both application energy and system energy consumption.

4 Application's Hardware Usage Analysis

In this section, we describe AppScope's techniques that are used to detect and analyze how each hardware component is used by an application.

4.1 Limitation of Previous Approaches

Conventional methods for estimating hardware component usage include HPCs, *procfs* and *sysfs* on Linux, and *BatteryStats* on Android. Each of these methods is limited in terms of their efficiency in application energy metering.

HPCs are a set of special registers that are built into microprocessors and are used to count certain processor events. These counters can be used for low-level performance evaluations or system tuning. With the use of HPCs, power consumption can be accurately analyzed. However, HPCs are highly dependent on a processor's architecture, and kernel modification is generally required to look into the HPC registers. Moreover, the counting results are effective only for CPU-and memory-related power analysis.

The Linux *procfs/sysfs* are special filesystems in Linux that provide information about processes, hardware usage, and other types of system information; *procfs/sysfs* are inadequate for monitoring application energy. First, the update rate of each hardware component is different, as is the data access method. For instance, with the Linux kernel 2.6.35.7 for Android Gingerbread, the update rate of CPU utilization is 5Hz and the CPU frequency is provided only for the current status. It is therefore difficult to decompose the CPU utilization of an application into each frequency. Also, due to the constraints in *procfs/sysfs* access, the application energy metering system should continuously poll both CPU utilization and frequency status to estimate CPU energy consumption. Second, the details of the information obtained from the filesystem vary depending on the type of underlying hardware. For example, WiFi traffic is not provided for process bases and GPS usage information is generally not available. Last, although the aforementioned limiting factors can be alleviated with kernel modification, the kernel should generally not be modified to support system portability on diverse platforms.

The Android *BatteryStats*, which provides battery status and hardware usage information, is a widely-used functionality for battery-related applications. *BatteryStats* inherits the fundamental limitations of *procfs/sysfs* and per-process usage information is not available for a certain type of hardware component. Furthermore, the granularity of information varies with hardware components. For example, *BatteryStats* produces component usage statistics on CPU and WNI traffic by reading *procfs/sysfs*, whereas display utilization is only available for the entire system.

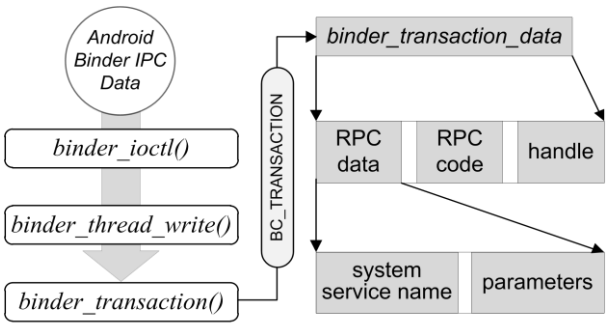


Figure 3: The Android IPC Data format for RPC procedure.

4.2 Kernel Activity Monitoring

Android applications typically access hardware components in two different ways. When an application uses hardware components supported by the Linux kernel, the application requests related system calls. Otherwise, application requests RPC via the Android binder [20, 21]. This section explains how AppScope uses the Android binder RPC mechanism to analyze component usage, and also how usage data are collected upon system calls.

4.2.1 Android Binder RPC

Android RPC is executed using binder RPC protocol, which is processed in the binder driver of the kernel. Figure 3 shows the data format of the Android IPC that is used for processing the BC_TRANSACTION command of the binder RPC procedure. To execute the stub interface of many service applications, the BC_TRANSACTION command is sent to the binder driver. At this moment, IPC data is sent to the binder driver with *binder_ioctl()*, and *binder_transaction()* executes the BC_TRANSACTION command within the binder driver. Thus, AppScope analyzes IPC data processed in *binder_transaction()* and collects data about the system usage. BC_TRANSACTION differentiates the requested functions using the RPC code of *binder_transaction_data*, as shown in Figure 3. The details of the requested command are known as "System Service Name" and "Function Input Parameter" within the RPC data.

4.2.2 Kprobes

Kprobes [22] is used to monitor the behavior of system calls. Kprobes is one of Linux's debugging mechanisms. It can dynamically insert break points during a kernel's runtime. It can be inserted into any kernel routine and

collect information non-destructively and without intruding into original kernel behavior. With this mechanism, the kernel function call can be monitored with low overhead because only a single instruction is substituted to detect the kernel operation. AppScope uses Kprobes to detect events on hardware component operations and to analyze a component's usage statistics. AppScope is compiled as a kernel module and controlled dynamically. Hence, apart from installing and removing the module, no additional user activity is required.

4.3 CPU Usage

In order to measure the consumed energy of process P_x, we need utilization u, as well as the CPU frequency relevant to u for a given time unit. In the Linux kernel, the utilization of P_x is computed using P_x's *utime() /stime()*. The *utime()/stime()* is estimated by detecting the switch from the TASK_RUNNING state to another one. Here, checking the states of all processes and updating their utilization for each scheduler call would generate a significant overhead. To reduce the overhead, AppScope detects the process switch by monitoring a wake-up event via *sched_switch()*. When a wake-up event occurs, AppScope updates the utilization of the previous process to calculate the utilization.

The CPU frequency changes according to the dynamic voltage and frequency scaling (DVFS) governor in the kernel. The *cpufreq_cpu_put()* function invokes a change in the frequency of the DVFS governor. Thus, the function is monitored and the frequency information is obtained at the call time. Frequency information, as well as information regarding system time, is then stored. Figure 4 illustrates the concept of the mechanism. Here, both the frequency change and the utilization value are computed based on the system time (jiffies), and each color indicates a separate process.

4.4 WiFi Usage

The energy consumption of WiFi varies according to the packet rate (i.e., transmitted packets per second). Thus, the amount of transmitted WiFi packets per given unit should be estimated to compute the energy consumption of process P_x. The data packet rate of process P_x depends not only on data size but also on protocol and maximum transmission unit (MTU). In our system, we have referred to the device agnostic network interface (DAI) layer of the Linux networking stack to estimate the packet rate. The DAI layer is an abstract layer located directly above the device driver layer (DDL), and it prepares (independently from the protocols) data for eventual transmission. In DAI, there are two main functions: *dev_queue_xmit()* for transmitted data and *netif_rx()* for received data. Figure 4 shows the WiFi

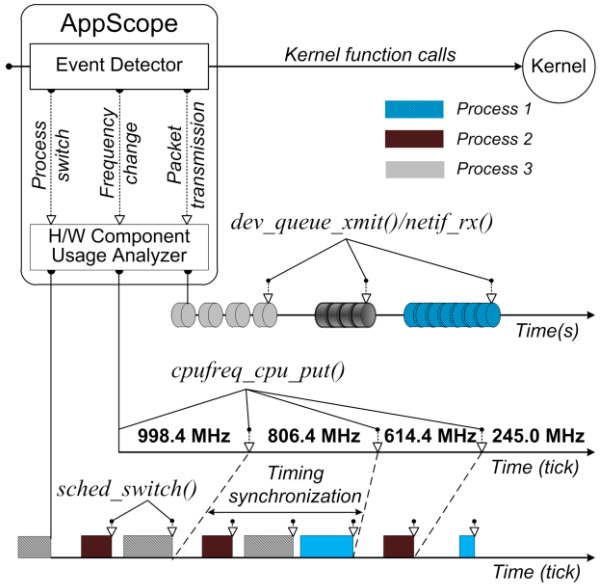


Figure 4: Analysis of CPU utilization/frequency, and the WiFi interface.

usage analysis of a process based on the detection of *dev_queue_xmit()* and *netif_rx()* calls. The packet rate is computed using the transmission/reception time of the packet. The power state is then identified based on the packet rate, and energy consumption is computed using activated time duration.

4.5 3G Usage

The energy consumption of a 3G interface depends on the RRC state. To efficiently utilize a radio resource in a 3G network, the RRC protocol typically defines three states: IDLE, FACH (forward access channel), and DCH (dedicated channel). Although the RRC state change depends on a carrier's policy, the RRC, in general, remains in the IDLE state when there is no data to send or receive. The state switches to the low power state, FACH, when data communication starts, and remains in the high power state, DCH, while data is being sent or received [23]. Our work is conducted in the Korea's SK-Telecom WCDMA network. In this network, mobile phones remain in the IDLE state if there is no data transmission. When data communication occurs, the mobile phone connects to the UMTS network for a short period of time, and then accesses the HSDPA network for a high-speed data transmission accompanying the RRC state transition. Thus, we identify the state transition of RRC based on the connection type of network.

The radio interface layer (RIL) daemon and vendor RIL of the Android telephony service are both located in the Linux user space. That is, voice calls and control commands are not processed using the Linux networking stack. Hence, 3G usage and the RRC state transition,

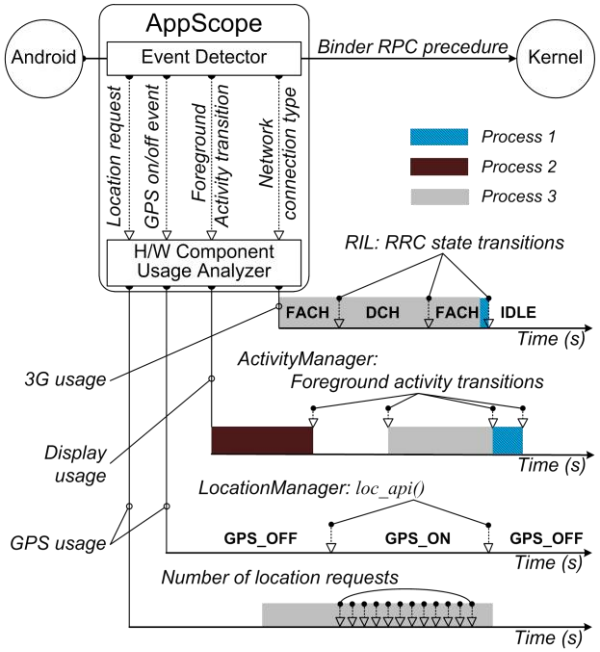


Figure 5: Analysis methods for 3G, GPS interface and LCD display usage information.

with the exception of data communication, cannot be analyzed within the Linux kernel. We therefore analyze the hardware component operation using the Android binder RPC. Figure 5 illustrates the concept of AppScope regarding 3G usage statistics. The change in network connection type is detected by checking the IPC data in Android RIL.

4.6 LCD Usage

The energy consumption of an LCD display is proportional to display brightness and display duration. Brightness can easily be identified from the current display settings of the Android framework. However, display usage, per application, cannot be directly obtained using the device routine within the kernel because the display operation is controlled by the Android framework. Therefore, AppScope recognizes foreground applications using the Android *ActivityManager* service, and its display usage is estimated by monitoring it. AppScope catches an event on foreground activity by checking the IPC data in the binder driver. When the process P_x's activity is in the foreground, display usage data is updated until another activity is brought into the foreground or the screen is turned off.

4.7 GPS Usage

The energy consumption of GPS is directly related to the power-on time of the interface. However, on/off time of a GPS system does not depend on the location request of the application. Also, several applications may simultaneously request location information from a GPS interface. Since the device interface for GPS is not exposed in the kernel, the estimation of process P_x's GPS usage is not trivial. In our work, we estimated cess P_x's usage statistics by monitoring *loc_api()* and *LocationManager* in the binder driver. The GPS interface is turned on/off with the *loc_api()*, and *LocationManager* provides location updates when GPS is turned on. Figure 5 illustrates how the AppScope estimates GPS usage of process P_x through monitoring the *LocationManager* of the Android framework. AppScope monitors *LocationManager* calls and calculates the GPS activation duration. During GPS activation, AppScope counts the location requests to *LocationManager*. The count is then used to estimate the energy consumption for each application process. Thus, when multiple processes request location information, AppScope distributes the energy consumption proportionally to the corresponding processes based on the usage count.

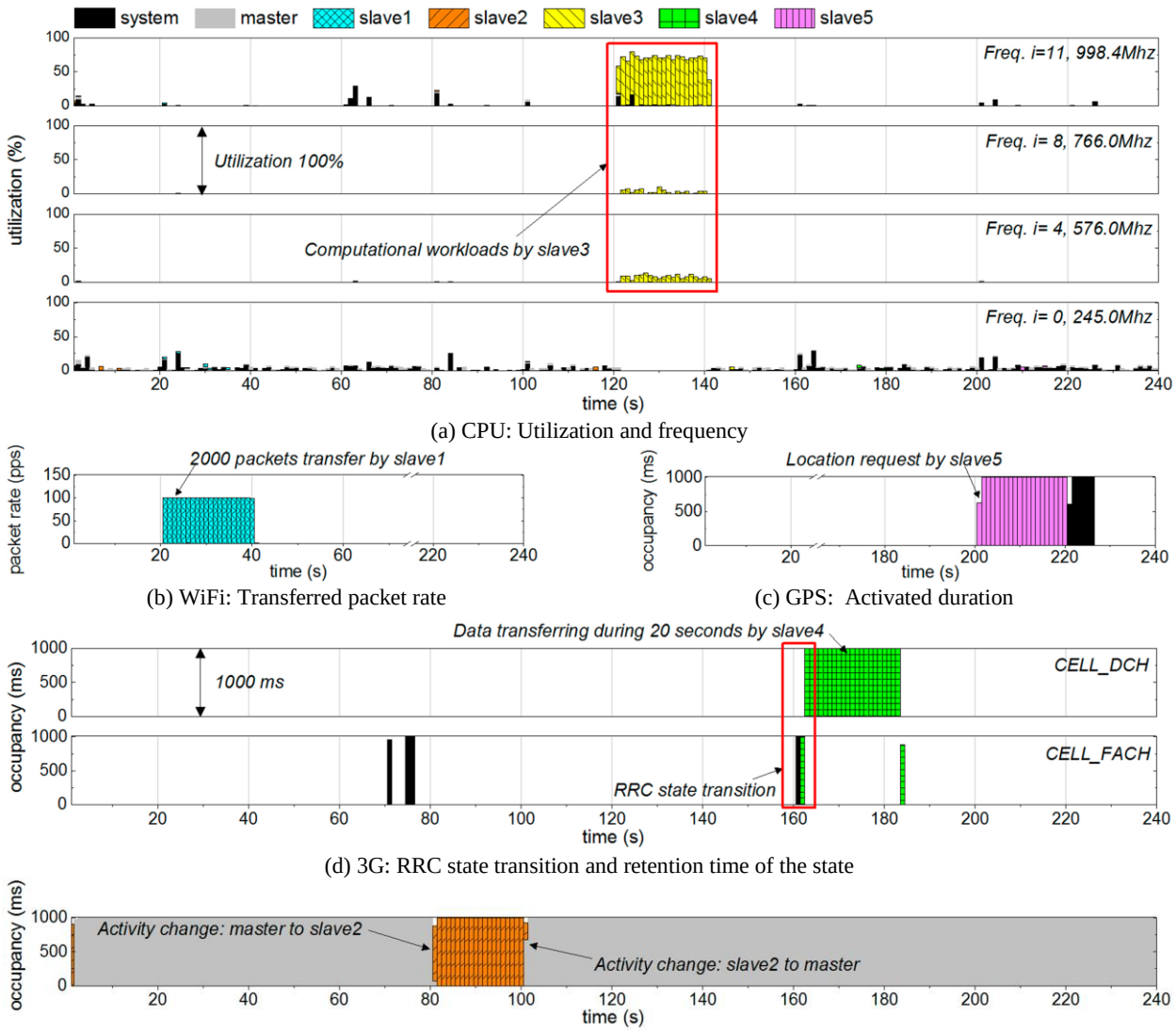
5 Evaluation

AppScope was developed in Linux kernel 2.6.35.7. The SystemTap version 1.3 [24] also uses Kprobes and data collection for the purpose of evaluation. All evaluations are carried out on HTC Google Nexus One (N1; Qualcomm QSD 8250 Snapdragon 1GHz, 3.7-inch Super LCD display) [25] with Android platform version 2.3. Note that N1 is equipped with a current sensor (MAXIM DS2784) upon which DevScope can build its power model. The Monsoon Power Monitor [26] is used as an external power meter.

In order to evaluate the AppScope framework, we

Table 2: Operation sequence of test applications

Time (sec)	Test App.	Operation	Description
0	Master	Run	Execution as a foreground activity and prevent screen off
20	Slave1	Transmit 2,000 packets via WiFi interface for 20 seconds	Approximate packet rates is 100pps
80	Slave2	Change the foreground activity for 20 seconds	After 20 seconds, Master app's activity return to foreground
120	Slave3	Start CPU job for 20 seconds	CPU frequency is changed by DVFS
160	Slave4	Transmit data via 3G interface for 20 seconds	RRC transition in the beginning of transmitting
200	Slave5	Turn on GPS interface for 20 seconds	Periodic updates of location information GPS



(a) CPU: Utilization and frequency

(b) WiFi: Transferred packet rate

(c) GPS: Activated duration

(d) 3G: RRC state transition and retention time of the state

(e) LCD: Foreground duration

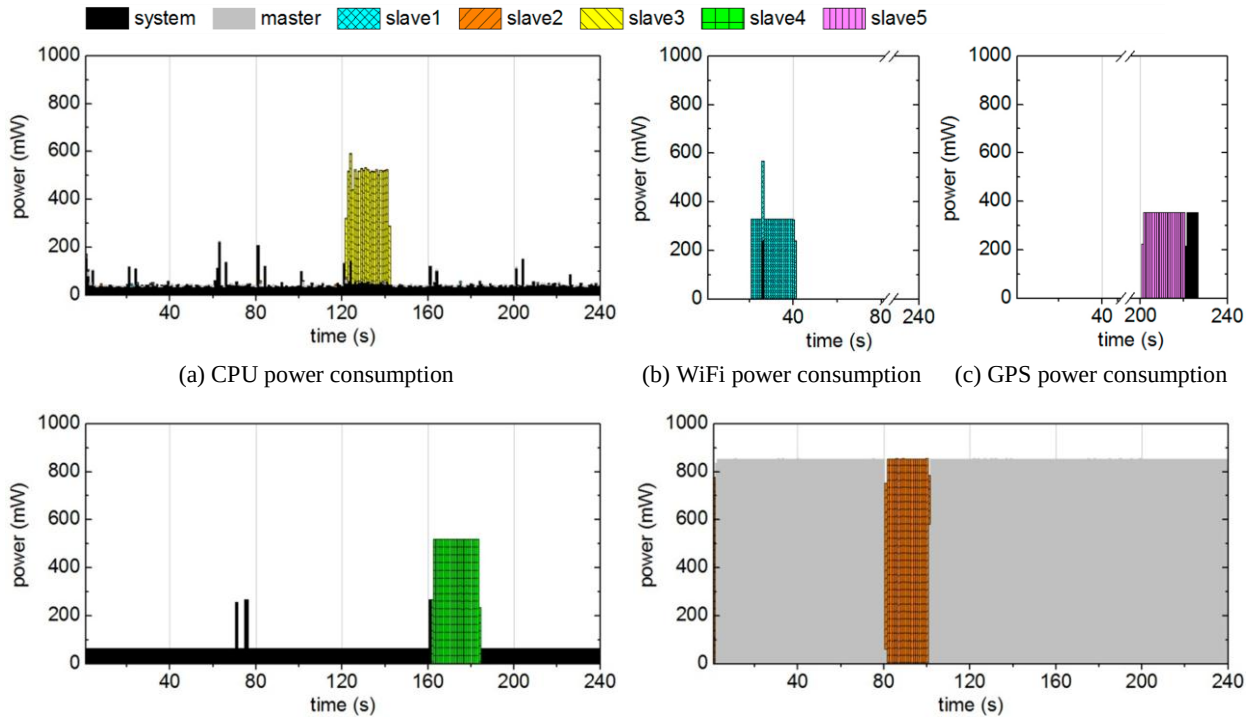
Figure 6: Hardware component usage trace of AppScope for test applications.

benchmarked a set of Android applications and estimated their energy consumption with AppScope. We also measured the overhead of AppScope in terms of power consumption and CPU utilization.

5.1 Component Usage Monitoring

To evaluate the accuracy of hardware event detection and collection of usage statistics, we designed and experimented on one "Master" and five "Slave" applications. The Master sets a pre-defined workload, executes the schedule of each hardware component workload, and controls the Slaves according to this schedule. We ran the Master and Slaves for 240 seconds in the order shown in Table 2.

Figure 6 shows the results of the tests on hardware component usage while running the test scenario in Table 3, where data was collected for every second. Each row in Figure 6(a) is differentiated by CPU frequency and *i* is the index in the frequency table for N1. The bar height represents utilization of relevant frequency. Due to space limitations in Figure 6(a), we have omitted some plots in which the utilization is too low or absent altogether. In the cases of GPS, LCD, and 3G, the power model requires activated time duration as usage information. The bar height in Figure 6 (c), (d), and (e) represents occupancy time (ms) in a unit time. The "system" stands for the system energy component, described in Section 3.3. The applications were started up at booting time and are differentiated by color.



(a) CPU power consumption (b) WiFi power consumption (c) GPS power consumption

(d) 3G interface power consumption (e) LCD display power consumption

Figure 7: AppScope power traces for test applications.

In Figure 6(a), from time *t*=121 for 20 seconds, Slave 3 actively utilizes the CPU. As a result, the CPU frequency increases, and the applications, which have been running in low frequency with low CPU utilization, started to operate in high frequency. Moreover, Slave 3 shows low CPU utilization in the same region with medium frequency due to the *OnDemand* policy.

In Figure 6(b), from time *t*=21 for 20 seconds, Slave 4 transmits data over WiFi. As shown in Table 2, Slave 1 sent 2,000 packets, i.e., its packet rate is 100pps. In Figure 6(c), Slave 5 uses GPS from *t*=201 for about 20 seconds. Note that after Slave 3 terminates the use of the GPS, the GPS is still used for about 4.5 seconds by the "system". With further experiments, we found that after an application terminates GPS usage, the "system" uses the GPS for a duration of about 2–4.5 seconds. After this timeframe, usage of the GPS interface is completely stopped.

Table 3: Power coefficient values of N1

Comp.	Index	Coefficient		Comp.	Index	Coefficient	
CPU	$freq$ *(Mhz)*	β_i^{freq}	β_i^{idle}	LCD	b	$\beta_b^{brightness}$	
	245.0	201.0	35.1		5	367.8	
	384.0	257.2	39.5		55	451.5	
	460.8	286.0	35.2		105	631.1	
	499.2	303.7	36.5		155	697.9	
	576.0	332.7	39.5		205	775.4	
	614.4	356.3	38.5		255	854.0	
	652.8	378.4	36.7	3G	rrc	β^{rrc}	
	691.2	400.3	39.6		IDLE	63.9	
	768.0	443.4	40.2		FACH	267.9	
	806.4	470.7	38.4		DCH	519.3	
	844.8	493.1	43.5	WiFi		β_l	β_h
	998.4	559.5	45.6		Transmit	1.2	0.8
GPS		β^{gps}			Base	238.7	247.0
	ON	354.7			Threshold	25pps	

In Figure 6(d) shows that the "system" transmits some data before Slave 4 transmits data at time *t*=161. After that, the RRC state remains in FACH for a short duration and changes to DCH. Also, as the packet transmission is terminated, the RRC state changes to FACH. This result is consistent with the RRC protocol between carrier and mobile devices on UMTS networks. Figure 6(e) shows the switching point for display between the two foreground activities. When Slave 2 activity is brought to the foreground, the display is not used by any of the applications for a duration of about 60–100ms. This is a blank duration when the activity change occurs in the *ActivityManager*.

In summary, AppScope detects hardware operation time as indicated in Table 2. The experimental results show that AppScope observes accurate usage of hardware components and correctly observes their power characteristics.

5.2 Energy Metering Validation

We estimate energy consumption for each application based on hardware component usage, as shown in Figure 6. In order to attain an accurate estimation, we used DevScope [19] to extract power coefficients (Table 3), based on the power model explained in Table 1 of Section 2.2. Note that all the experiments for communication interfaces, such as WiFi, 3G, and GPS, were conducted at a stationary place, i.e., fixed-strength radio signals; hence, we did not consider energy effects on varying signal strength for these components. We compared the estimation results with the results obtained from the Monsoon power meter.

5.2.1 Granularity

Figure 7 shows the power consumption of hardware components per application. Figure 7(a) shows the CPU power consumption for the entire duration – 240 seconds. Overall, the "system" uniformly consumed approximately 100mW. The power consumption of Slave 3 is about 480 mW in the increased frequency region. As shown in Figure 7(b), (c), and (d), when communication components, such as WiFi, 3G, and GPS are used, we observed that the "system" consumes a certain amount of power. In Figure 7(e), when the application uses an LCD display, the power consumption of the LCD is relatively higher than other components. Master consumed the highest energy due to long display occupancy. However, it did not operate other hardware components. Table 4 shows the estimated energy results by aggregating the results shown in Figure 7. As shown in Table 4, AppScope provides application-specific energy consumption data for each hardware component, even when multiple applications run in parallel.

5.2.2 Accuracy

To analyze the correctness of energy consumption results obtained in Section 5.2.1, we compared our results with those obtained using the Monsoon power meter. Figure 8 shows the comparative results between AppScope estimation and external measurement. The aggregated power consumption of all applications using AppScope is similar to the entire power consumption measured using the external power meter. However, a power difference of about 100–400mW has been observed in some regions. At time t=60 for 10 seconds, the external measurement showed that power consumption temporally increased for a short period of time. This is because Slave 4 turned off the WiFi interface and the "system" automatically activated the 3G interface. In this process, AppScope noticed that the "system" sent packets over the 3G interface, but the 3G interface's power consumption was not detected due to the WiFi's turn off delay and 3G interface activation. When the CPU frequency rises, a large difference exists between the external measurement result and the power consumption estimated by AppScope. At time t=120 for 20 seconds, power consumption increases due to the CPU frequency and increased utilization. At this moment, the power consumption was estimated as 1400mW, which is 7% less than the external measurement result. This demonstrates the limitations of our simple CPU power model, which ignores the effects of cache, bus, memory and other SoC components. More accurate models can be built by using performance counters to account for these effects [5, 10-13]. Figure 8 summarizes that the overall energy consumption estimated by Monsoon is 282.8J, and 268.0J by AppScope, which is a 14.8J (5.2%) difference.

Table 4: Energy estimation of test applications

App.	CPU(J)	WiFi (J)	GPS (J)	3G(J)	LCD(J)	Total(J)
System	11.3	0.2	2.0	14.7	0	28.2
Master	0.5	0	0	0	186.8	187.3
Slave 1	0.04	6.80	0	0	0	6.84
Slave 2	0.1	0	0	0	17.9	18.0
Slave 3	9.3	0	0	0	0	9.3
Slave 4	0.01	0	0	11.41	0	11.42
Slave 5	0.01	0	7.00	0	0	7.01

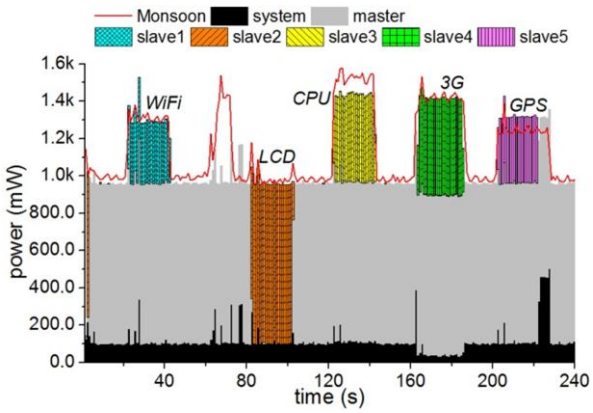


Figure 8: AppScope results vs. Monsoon measurement results for test applications.

5.3 Overhead Analysis

To estimate the overhead of AppScope, we have performed the experiment described in Section 5.1 by loading and unloading AppScope onto the system. In both scenarios, power consumption is estimated using the Monsoon power meter. Figure 9 shows the results. During the experiments, test applications occupied the display activity. Therefore, the information regarding

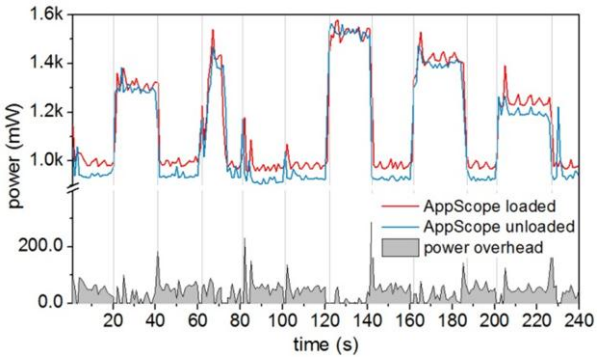


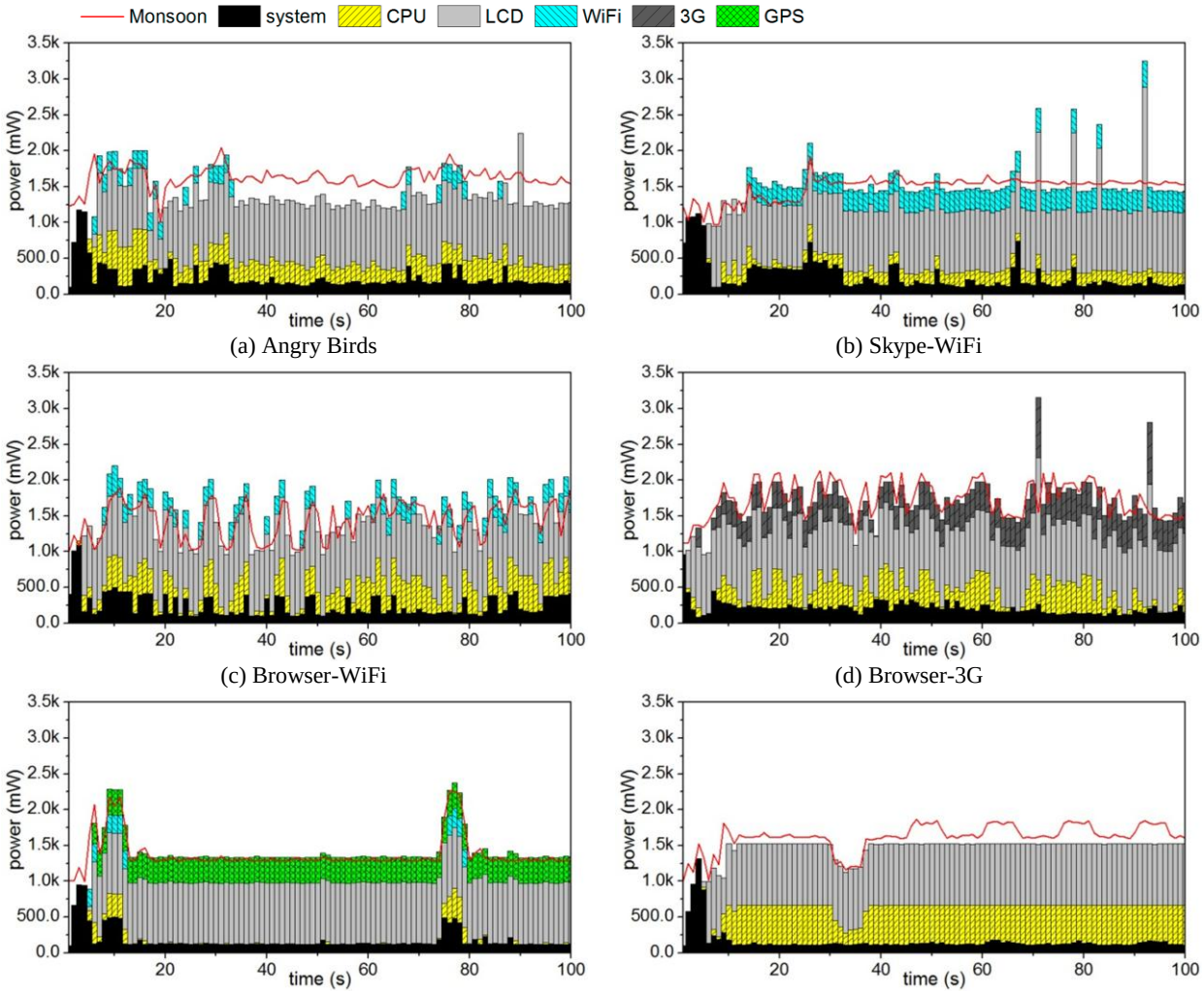
Figure 9: Overhead analysis of AppScope.

the power consumption of displays (Figure 9) was collected during the entire duration of the experiment. During CPU testing, the power consumption does not increase between 120-second to 140-second (see Table 2). While WiFi and 3G tests are carried out, the energy consumption slightly increases in comparison to the energy consumption experienced with the display only function. Within 240 seconds, AppScope generated 8.4J energy overhead, which is a 34.9mW increase on average. Moreover, the five tests showed that AppScope generated 2.1% CPU overhead on average, with a standard deviation of 1.9 and the worst case being 5.9%.

AppScope is a Linux kernel module and can be dynamically loaded/unloaded at runtime. Thus, users may install AppScope when analysis is required and remove it if unnecessary. Consequently, when AppScope is not activated, the overhead is not generated at all.

6 Real Application Energy Metering

We have evaluated AppScope's energy metering performance using applications distributed via *Google Android Market*. For this analysis, we have selected four applications that adequately utilize each component.



(a) Angry Birds

(b) Skype-WiFi

(c) Browser-WiFi

(d) Browser-3G

(e) Google Maps

(f) StabilityTest Benchmark

Figure 10: AppScope power traces for real applications.

Figure 10 shows the estimated energy consumption of Angry Birds (game), Skype (VoIP), web browser, and Google Maps (location provider). Energy consumption of a web browser is divided into two cases, i.e., browsing with WiFi or 3G. To compare the estimated results using the Monsoon power meter, we also show the energy consumption of the system and applications, per component energy consumption.

AppScope showed accurate estimation results in comparison to the external measurement results. As shown in Figure 10(a), the power consumption of Angry Birds showed the highest error among the five test cases. Specifically, the uniform amount of 300mW error was shown (except for the WiFi period) after the game is completely loaded, i.e. 20 seconds. CPU and LCD were continuously used in the region where high error is shown. Compared to the other four cases, we understand that the game activates N1's GPU (Integrated Graphics Processing Unit Adreno 200 on Qualcomm QSD8250 Snapdragon) and error is caused by this hardware component. To find out the exact cause of the error, we have conducted additional experiments with Android's CPU/GPU benchmark tool, StablilityTest [27]. As illustrated in Figure 10(f), while StablilityTest is preparing 3D objects to display on the screen (initial 37 seconds), the AppScope results and the results of the external power meter were nearly identical. Between 40 to 100-seconds where a 3D object was periodically rotated, there was a 300mW difference. The difference is as large as the error shown in Figure 10(a). In this region, CPU utilization was 100%. Hence, the error is assumed to be caused by the GPU operation.

In Figure 10(b) and (c), the power consumption of the WiFi interface was reflected in total energy consumption with approximately 4% error. Note that the Monsoon results in Figure 10(b) are higher than AppScope, whereas the results are opposite in Figure 10(c). We consider this to be a limitation of the linear regression-based power model that is produced by DevScope. Although the WiFi interface always operates with CPU, our power model does not consider inter-dependency of WiFi interface and CPU. This limitation may be overcome using a model that includes cross-terms, which represent the inter-dependency among components [5].

As shown in figures 10(a), (b), and (d), after 70 seconds of operation, there was a temporary increase in the energy consumption of LCD and 3G interface. These increases are generated due to an error in the data collecting program, which was implemented using SystemTap [24]. In these points, the collected workload for 2 seconds is accumulated in 1 second by a timer bug in SystemTap. In reality, there should not be a temporary increase in power consumption of LCD unless its brightness is changed. After the increase in power consumption, there was a time difference in the estimation of AppScope and Monsoon measurement.

Table 5: Energy estimation for real applications

App.	CPU (J)	WiFi (J)	GPS (J)	3G (J)	LCD (J)	System (J)	Total (J)	Monsoon (J)	Err. (%)
Angry Birds	27.4	7.1	0	0	80.3	24.0	138.8	162.7	14.7
Browser (WiFi)	28.6	14.3	0	0	82.8	25.1	150.8	144.3	4.5
Browser (3G)	25.7	0	0	36.2	85.9	13.3	161.1	174.1	7.5
Skype (WiFi)	14.8	24.6	0	0	85.0	25.7	150.1	148.8	0.9
Google Map	3.9	2.5	33.6	0	81.2	18.0	139.2	137.8	1.0

Table 5 shows each application's total and component-wise energy consumption. The total energy consumption is computed by aggregating the energy consumption of the hardware components and the system. The error is calculated using total estimated energy consumption and the results from the external power meter. All applications, with the exception of Angry Birds, showed an error rate below 7.5% during a 100-second experiment. Angry Birds showed a 14.7% error due to the aforementioned GPU operation.

7 Related Work

Recent research [7, 9, 14, 16] on smartphone power management has developed diverse power models to estimate a device's power consumption. Dong and Zhong proposed Sesame [16], which is an automatic smartphone power modeling scheme using a built-in current sensor. Their work focused on overall system power rather than power analysis on individual hardware components. This feature is hardly applicable for estimating the energy consumption of each application. Pathak et al. [14] proposed an FSM (finite state machine)-based power model using an external power measurement tool in conjunction with system call tracing. This approach may be applicable for application energy metering, but in-depth study and measurements on target devices should be required to obtain detailed power states.

Among recent works, PowerTutor [7, 8], PowerProf [9], and Eprof [29] support the estimation of application energy consumption. PowerTutor [8] is an application power estimation system that uses PowerBooter [7], which is a power model generation tool using fuel gauge sensors and knowledge of battery discharge behavior. PowerTutor [8] uses different methods to access usage statistics from *procfs* and *BatteryStat* for each hardware component. This method cannot guarantee the accuracy

of application energy consumption, due to the limitations that are discussed in Section 4.1. PowerTutor provides UID-specific energy information, but not process-specific information. Furthermore, it requires modification of the Android system software and kernel for components such as GPS and Audio. With AppScope, we use standard kernel functionalities to collect hardware usage information through an event-driven mechanism; this avoids monitoring overhead and performance degradation. In addition, AppScope provides process-specific power estimation in real-time.

Kjærgaard and Blunck proposed PowerProf [9], which is an unsupervised power profiling scheme for the smartphone using the Nokia Energy Profiler [28]. PowerProf generates component power models based on a genetic algorithm in order to automatically identify the power states of underlying hardware components. PowerProf enables online energy estimation, but the scheme is focused on power modeling rather than application energy metering. PowerProf measures power consumption for API calls issued in programming language. This method is limited in terms of application energy metering because the technique strongly depends on the programmer's intention.

Eprof [29] is a fine-grained energy profiler for smartphone applications. Based on the FSM power model [14], Eprof has the ability to analyze the asynchronous energy state of an application, modeling the tail-state energy characteristics of hardware components with routine-level granularity. Energy metering is achieved via a post-processing mechanism using an explicit accounting policy. Eprof requires modifications in the Android framework to trace the API calls; the application code, if using the Android NDK, should also be modified.

PowerScope [6] and Quanto [30] are developed towards energy estimation with hardware usage monitoring. PowerScope [6] provides detailed process-specific energy estimation for mobile devices. The scheme requires an additional computing resource, and programmers should use a set of specialized APIs to estimate power consumption. Quanto [30] is developed as a network-wide energy profiler for fast energy metering based on event-driven methods in TinyOS. The approach is similar to AppScope, which detects hardware operations in kernel, and breaks down the energy usage of a system by hardware component.

The information obtained with AppScope is closely related to energy efficient operating system research [1-5]. These works, in fact, proposed abstract OS mechanisms to limit energy that can be used by processes. The mechanism requires usage and energy consumption information regarding an application's hardware. In this context, AppScope would be useful for developing energy-aware operating systems.

8 Discussion

The accuracy of application energy metering depends on the power model of underlying hardware components. The present work used the power model of DevScope, which currently does not cope with GPU, multi-core, and memory components. Indeed, the experimental results in Figure 9(a) showed that a relatively large error is exhibited in applications using the integrated GPU. In addition to the GPU, recent smartphones are beginning to employ multi-core CPU, which necessitates the development of more advanced tools covering new hardware features. Also, the current AppScope/DevScope is limited in modeling the memory hardware component. In fact, previous work [15] showed that energy characteristics of smartphone applications differ with the nature of application; that is, CPU-bound or memory-bound jobs. We are aware that in order to model diverse hardware and obtain applications' energy consumption more accurately, both AppScope and DevScope should be supplemented with further emphasis on memory, GPU, and multi-core CPU architecture. This is, in fact, part of our future work.

Meanwhile, the tail-state energy consumption of cellular, WiFi, and GPS hardware components should be considered for fine-grained energy modeling. The Finite States Machine (FSM)-based model [14], for instance, uses power state transitions, instead of component utilization for power modeling, which enables the accurate modeling of tail-state. AppScope, however, does not detect the tail-state; hence the energy consumption on this state is not reflected in the application's energy. This limitation is fundamentally caused by the use of a linear power model in AppScope, which primarily obtains usage statistics, rather than state changes, of individual hardware components.

Although the AppScope energy metering framework includes DevScope as its core component to obtain device power models automatically and online, the core part of the AppScope framework is still the automatic acquisition of x_i^j and d_i^j for each hardware component accessed by an application. This means that the core of AppScope can practically run on any smartphone whose component power models are known *a priori* - either by DevScope or by direct measurement of individual hardware components.

9 Conclusion

In this paper, we proposed AppScope to automatically meter energy consumption of Android applications using kernel activity monitoring. AppScope traces system

calls and also analyzes Android binder IPC data. Designed as a kernel module, AppScope runs efficiently to collect fine-grained process-specific energy information. Compared to previous research on smartphone energy estimation, AppScope provides a more accurate and detailed application-specific energy estimation solution. This result will be used as an important basis in establishing a foundation to support power-related research on Android mobile devices.

Acknowledgements

We would like to thank the anonymous reviewers for their comments. A special thank you should go to our shepherd, Gernot Heiser, who has greatly helped us enhance the quality of this paper. We also appreciate the comments from Rodrigo Fonseca. 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.2011-0015332).

References

[1] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing Energy as a First Class Operating System Resource, *ACM SIGPLAN Notices*, volume 37, pages 123–132, 2002.

[2] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Currentcy: A Unifying Abstraction for Expressing Energy Management Policies. In *USENIX ATC*, 2003.

[3] H. Zeng, C. Ellis, and A. Lebeck. Experiences in Managing Energy with Ecosystem. *IEEE Pervasive Computing*, 4(1):62–68, 2005.

[4] A. Roy, S. Rumble, R. Stutsman, P. Levis, D. Mazi ères, and N. Zeldovich. Energy Management in Mobile Devices with the Cinder Operating Systemg. In *EuroSys*, 2011.

[5] D. Snowdon, E. Le Sueur, S. Petters, and G. Heiser. Koala: A Platform for OS-level Power Management. In *EuroSys*, 2009.

[6] J. Flinn and M. Satyanarayanan. Powerscope: A Tool for Profiling the Energy Usage of Mobile Applications. In *IEEE WMCSA*, 1999.

[7] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior based Power Model Generation for Smartphones. In *CODES+ISSS*, 2010.

[8] Powertutor, http://powertutor.org.

[9] M. B. Kjærgaard and H. Blunck. Unsupervised Power Profiling for Mobile Devices. In *Mobiquitous*, 2011

[10] Y. Xiao, R. Bhaumik, Z. Yang, M. Siekkinen, P. Savolainen, and A. Yla-Jaaski. A System-level Model for Runtime Power Estimation on Mobile Devices. In *GreenCom-CPSCom*, 2010.

[11] T. Li and L. John. Run-time Modeling and Estimation of Operating System Power Consumption. *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 160–171, 2003.

[12] S. Gurun and C. Krintz. A Run-time, Feedback-based Energy Estimation Model for Embedded Devices. In *CODES+ISSS*, 2006.

[13] K. Singh, M. Bhadauria, and S. McKee. Real Time Power Estimation and Thread Scheduling via Performance Counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[14] A. Pathak, Y. Hu, M. Zhang, P. Bahl, and Y. Wang. Fine-grained Power Modeling for Smartphones Using System Call Tracing. In *EuroSys*, 2011.

[15] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX ATC*, 2010.

[16] M. Dong and L. Zhong. Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems. In *MobiSys*, 2011.

[17] Batterydiviner, https://play.google.com/store/search?q=batterydiviner.

[18] J. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. Snoeren, and R. Gupta. Evaluating the Effectiveness of Model-based Power Characterization. In *USENIX ATC*, 2011.

[19] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha. Non-intrusive and online power analysis for smartphone hardware components. *Technical Report*, MOBED-TR-2012-1, Yonsei University, 2012.

[20] Openbinder, http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderOverview.html.

[21] Android Binder, https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf.

[22] Kprobes, http://www.kernel.org/doc/Documentation/kprobes.txt.

[23] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: A Cross-layer Approach. In *Mobisys*, 2011.

[24] SystemTap, http://sourceware.org/systemtap.

[25] HTC Google Nexus One, http://en.wikipedia.org/wiki/Nexus_One.

[26] Monsoon, http://www.msoon.com/LabEquipment/PowerMonitor.

[27] StabilityTest, https://play.google.com/store/apps/details?id=com.into.stability.

[28] Nokia Energy Profiler, http://www.developer.nokia.com/Resources/Tools_and_downloads/Other/Nokia_Energy_Profiler/Quick_start.xhtml.

[29] A. Pathak, Y. C. Hu, and Ming Zhang. Fine Grained Energy Accounting on smartphones with Eprof. In *EuroSys*, 2012.

[30] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *USENIX OSDI*, 2008.