# FEPMA: Fine-Grained Event-Driven Power Meter for Android Smartphones Based on Device Driver Layer Event Monitoring*

Kitae Kim[1], Donghwa Shin[2], Qing Xie[3], Yanzhi Wang[3], Massoud Pedram[3], and Naehyuck Chang[1]
[1]Seoul National University, Korea, [2]Politecnico di Torino, Italy, [3]University of Southern California, USA
[1]{ktkim, naehyuck}@elpl.snu.ac.kr, [2]donghwa.shin@polito.it, [3]{xqing, yanzhiwa, pedram}@usc.edu

***Abstract*—This paper introduces a novel sensor-less, event-driven power analysis framework called FEPMA for providing highly accurate and nearly instantaneous estimates of power dissipation in an Android smartphone. The key idea is to collect and correctly record various events of interest within a smartphone as applications are running on the application processor within it. This is in turn done by instrumenting the Android operating system to provide information about power/performance state changes of various smartphone components at the lowest layer of the kernel to avoid time stamping delays and component state observability issues. This technique then enables one to perform fine-grained (in time and space) power metering in the smartphone. Experimental results show significant accuracy improvement compared to previous approaches and good fidelity with respect to actual current measurements. The estimation error of the proposed method is lower by a factor of two than the state-of-the-art method.**

## I. Introduction

Power consumption of smartphones is increasing with each generation of new devices. This power increase is caused by the need to provide more functionality, higher performance, ultra high resolution displays, high-speed wireless communication, etc. The capacity of batteries that power up such devices is also increasing, albeit at a much lower pace. This gives rise to the need to reduce power consumption of smartphones without limiting the functionality or curbing the performance. This is a very challenging undertaking considering the functional and performance requirements.

In this paper, we will use the term smartphone component to refer to all onboard modules in a commercial smartphone such as the application processor (AP), Wi-Fi, Bluetooth, cellular, GPS, cameras, display, flash memory, etc. Clearly, it is essential for the success of any system-wide power management solution to have an accurate accounting of which components within the smartphone are consuming power at a given time instance by how much. Power minimization in smartphones is typically done through some kind of power management solution whereby the unused components are power gated (turned off) while the power and performance level of active components is reduced to meet performance requirements. Other techniques such as scheduling tasks to maximize component idle time so that they can be put to sleep and offloading compute-intensive tasks to the cloud have also been suggested [1], [2], [3].

*This research is supported by the International Research & Development Program and the Center for Integrated Smart Sensors as Global Frontier Project of the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (No.2013075022, CISS-201373718). The ICT at Seoul National University provides research facilities for this study.

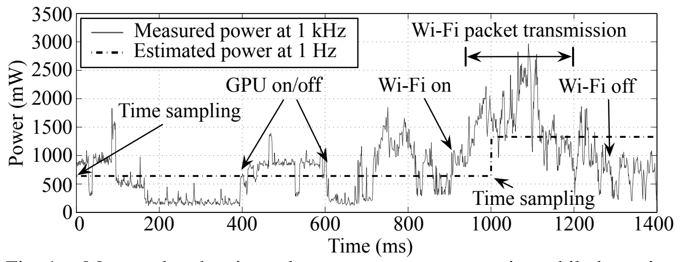

Fig. 1. Measured and estimated system power consumption while browsing a website using the Google Chrome browser. The system power consumption is measured by a DAQ at 1 kHz sampling frequency, and the estimated power is predicted by a sensor-less and sampling-based power estimation method that has 1 Hz sampling rate. The 1 Hz estimation does not detect GPU on/off during 400 ms and 600 ms and perceive the Wi-Fi packet transmission start at 950 ms and the end at 1200 ms.

Using current sensors for each component can provide highly accurate power measurements, which show how much power each component is consuming instantaneously, in real time. Unfortunately, the stringent form factors, weight, and cost constraints for smartphones prevent a manufacturer from employing the current sensor for each component. Instead, indirect power measurements, which estimate the power consumption of smartphones, are widely used. These power estimation techniques typically rely on offline, but state-dependent power characterization of components augmented by dynamically obtained information about the state of each component (e.g., active, idle, and sleep) and activity level (e.g., activity factor in an AP and packet transmission rate for a wireless link). In order to obtain the latter information, it is important to have the right system software support. Critical to the success of the aforesaid approach is the ability to correctly and quickly collect information about the state of each component such as the operating state of the Bluetooth module, utilization level of the AP, or data communication rate of the Wi-Fi chipset in an asynchronous (event-driven) manner. However, this ability is hard to come by because it requires operating system (OS) kernel support and fine-grained data collection of individual components with high timing accuracy in the smartphone. If these conditions are not met, indirect sensor-less power metering will become inaccurate.

Fig. 1 shows actually measured (1 kHz sampling) and estimated (1 Hz sampling) power dissipation profiles of a smartphone. The figure and its caption explain the power estimation challenge. In contrast, we will show a fine-grained event-driven indirect power estimation to capture most of the important events in a timely manner without aliasing. A further benefit of the fine-grained event-driven power estimation is that it can be used to guide dynamic power management (DPM) techniques for a smartphone. This is motivated by the fact that modern smartphone systems exhibit a rather short

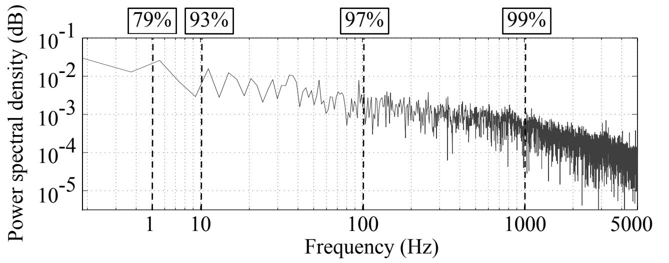

Fig. 2. Power spectrum of a smartphone load current and the energy distribution by the frequency.

idle period as illustrated in Fig. 1. Using the sampling-based approach for determining the state of a component does not capture the device state change in a timely manner, which can inhibit proper deployment of effective system-wide DPM policies. Nevertheless, previous smartphone power estimation methods sample the component states in the order of once every second or so [2], [4], [5], [6]. This is an extremely low sampling rate for modern digital systems. Even if the authors knew about this shortcoming, they could not help it because using a sampling rate of even 10 Hz in their power metering frameworks would not be practical due to rapid increase in the profiling overhead as well as timing errors in recording the events of interest.

The key to overcome limitations of previous approaches is twofold:

1) Avoid a regular sampling strategy and instead adopt event-driven recording of component state changes [7], [8]. This is a well-known technique not to lose time granularity while maintaining reasonable profiling overhead. However, there is a critical condition that should be satisfied in the event-driven approach. Events should be captured at a correct position in the OS event propagation flow.

2) Capture and record events of interest at the lowest level of the OS kernel so that one can avoid time stamping delays and state observability issues caused by capturing events at improper positions. We remind the reader that there are many different positions to capture device state change events in the propagation flow. Modern smartphones are equipped with a full-featured OS that has lots of software locks and memory buffers from a user program to hardware devices. An event (say a command to transmit data packets via the Wi-Fi module) that arrives at the OS kernel entry point can be delayed before it actually gets to the hardware component due to the locks and buffers, and vice versa. Hence, if the state changes of smartphone components are monitored at the highest level of the OS kernel, the time recording of these events can be far from reality of what happens in the device itself.

In theory and practice, it is not possible to achieve highly accurate instantaneous power estimation by sampling at a low rate or event monitoring at improper positions in the event propagation flow. Moreover, a comparison of estimation accuracy should be made with a golden reference that correctly captures the actual power dissipation profile. In order to do this, we measure the load current of a commercial smartphone, which is Samsung Galaxy Nexus, using a high-speed data acquisition system (DAQ) at a 50 kHz sampling frequency and visualize its power spectrum in Fig. 2. This figures shows that even a 200 Hz sampling frequency has a *temporal aliasing* because 3% of frequency components in the power spectrum are distributed above it. Since the golden reference contains most important parts of the power spectrum, we conclude that the sampling rate for the golden reference should be at least 1 kHz.

## II. Sensor-less Power Metering For Smartphones

The power consumption of a smartphone largely depends on application and user behaviors, which in turn determine component-level hardware activities and power modes. Sensor-less power metering methods read the necessary information such as system activities and device power modes to predict the instantaneous power consumption. Previous works on sensor-less power metering for smartphones can be classified into two categories: i) sampling-based methods, ii) event-driven methods.

Sampling-based methods periodically collect the information of hardware activities and power modes from the Linux kernel and predict the system power consumption based on these information. The regression-based power model [2] estimates power consumption by logging user activities. PowerBooter [4] provides an online power model generation technique using the battery discharge curve. The model molding technique [5] is presented to enhance the battery monitoring-based methods. However, a common issue that all sampling-based techniques face is the excessive performance and energy overhead due to frequent accessing the Linux kernel and reading large amount of data from the kernel. In addition, authors of [5] rely on a smart battery interface with a fuel gauge that can measure battery voltage, temperature, and even current sometimes. However, the sampling rate of the battery fuel gauge is too low to provide accurate results.

In contrast, event-driven methods collect system activities only when an event, which affects the system power consumption, occurs. This nature drastically alleviates the profiling overhead that makes it more suitable for the sensor-less power metering. The simplest way for implementing the event-driven method is modifying the source codes of Android or the Linux kernel. A fine-grained energy profiler called Eprof [7] predicts the power consumption by modifying the kernel code and accounting system call events. Authors of [8] develop a non-disruptive method called AppScope. They use the debugging tools of the Linux kernel to collect the information by setting a breakpoint to an arbitrary kernel function without any code modification. However, these event-driven approaches also have drawbacks such as inaccurate event timestamps, unobservable devices, and low time granularity of the power metering. Moreover, they do not take account of a multi-core processor and an organic light emitting diode (OLED) display in their power models. These drawbacks hinder the accurate power metering and thus limit the proper DPM for smartphones.

## III. Fine-grained Event-driven Power Metering Framework

### A. Device Driver Layer Event Monitoring

The Linux kernel consists of three important parts: the system call interface, the abstracted generic kernel layer, and the hardware-dependent device drivers. As the name implies, the system call interface is an interface between user-space applications and the generic kernel layer. The abstracted generic kernel layer is comprised of kernel routines that are independent from specific hardware components. The device drivers under the generic kernel layer are hardware-dependent routines for each hardware component that the kernel supports. Previous researches [7], [8] monitor system events on the abstracted generic kernel layer. However, their approaches are not accurate as they miss some events or collect inaccurate events. This is due to the fact that the abstracted generic kernel layer does not directly communicate with the hardware components.

In contrast, we directly monitor the events in the device driver layer. We adopt a non-disruptive debugging tool and define customized callback handler functions for the tool in the device driver layer. For each device component, we program the callback handlers to monitor events of interest such that system activities or the power mode change. We also record the timestamp of each event as well. Based on these events, we obtain the necessary information such as the operating status (i.e., whether the device is being power gated or clock gated), frequency, utilization time, and so on. These information are used in corresponding power model to obtain the power consumption for each component. Because power modes and utilization for each component are all controlled by their driver, monitoring the events in the device driver layer has two advantages as follows against previous approaches.

First, monitoring the events in the device driver layer provides the most accurate utilization time for hardware components. For example, in the network-related components, a significant time lag exists between a logged event timestamp at the generic kernel layer and actual time of the event at the device driver layer. This actually becomes one of main reasons why previous approaches did not yield accurate results in fine granularity. In fact, there are several buffers between the TCP/IP protocol stack of the generic kernel layer and the device driver of a network interface card (NIC). During a packet transmission, transmission-related events are captured immediately after sending packets are inserted into a socket buffer in the TCP/IP stack if we monitor the events at the generic kernel layer. However, the packets are not sent out until they move to the bottom of a queue in the NIC device driver. Therefore, packets in the buffer and the queue can be delayed during the transmission depending on the current system status and Linux kernel's policy.

Second, monitoring the events in the device driver layer allows us to access more devices. As the abstracted generic kernel layer communicates with the underlying device drivers through pre-defined interfaces, it does not know the detail of specific hardware operations in the device drivers. Therefore, the existing techniques [7], [8] are unable to capture some events from several hardware components such as the GPU, digital signal processor (DSP), and GPS. For example, there is no source code to control activity of the GPS module in the generic kernel layer. Although the authors of [8] manage to overcome the GPS issue by collecting events from the Android framework, it is less accurate compared to doing it in the device driver layer. By monitoring the events in the device driver layer, we have access to all of these components aforementioned.

## B. Power Modeling of Major Components

In this paper, we use Samsung Galaxy Nexus, which is an Android smartphone developed by Google and Samsung in October 2011, as the target smartphone platform. We identify several power consuming components in the target platform including the multi-core processor, Wi-Fi, cellular, GPS, and display-related modules. For each component, we present a power model based on the accessible information such as the frequency, utilization, power modes, and so on. Note that these components are "logical" in the sense that the CPU and GPU may be physically integrated on the same AP chip, but they are typically activated at different time when running different applications. Thus, we are able to report the power consumption of CPU separately from that of the GPU although they are on the same physical chip. In addition, the power consumption of other supporting circuitry when the CPU is active will be reported as the CPU power. For example, memory power is included in the CPU power report because these two physical components are logically working together when the CPU is active.

*1) Multi-core Processor:* The processor core is one of the major power consumers in the smartphone [9]. Common APs include multiple processor cores in the system-on-chip (SoC) design of an AP to provide the computational and processing power of state-of-the-art smartphones. Unfortunately, previous researches focusing on power characterization and modeling of smartphones [2], [4], [6], [8] have only considered a simple single-core processor. As a result, these techniques become inaccurate when multiple applications and/or multiple threads of the same application are running on the multi-core processor in the state-of-the-art smartphones.

An intuitive way to build a multi-core power model is to linearly scale up the single-core power model by the number of cores. However, we observe that the power consumption of a dual-core processor in the target platform does not increase linearly with the number of active cores. Thus, we characterize the dual-core power consumption as a function of the operating frequency, utilization, and normalized co-utilization rate of each core. The utilization of each core is the amount of time that the core has spent in user and kernel mode for handling a process. The normalized co-utilization rate of each core that lies between 0 and 1 is a ratio that the core has been concurrently used with another core in the utilization time, and it is calculated through dividing the concurrent utilization time by the total utilization time of the core.

The proposed method monitors events related with a change of the operating frequency, utilization, and suspend state of each core. We apply different power models depending on the current status of the dual-core processor as follows: i) if the dual-core processor is not in suspend mode, and the total utilization of all cores is higher than zero,

$$P_{CPU} = \frac{1}{\Delta t} \sum_{i=1}^{n} \left[ r_i \; u_i^{cpu} \; \beta_{dual[freq]}^{cpu} + (1 - r_i) \; u_i^{cpu} \; \beta_{single[freq]}^{cpu} \right], \tag{1}$$

ii) if the dual-core processor is not in suspend mode, but the total utilization of all cores is zero,

$$P_{CPU} = \beta_{idle[freq]}^{cpu}, \tag{2}$$

iii) if the dual-core processor is in suspend mode,

$$P_{CPU} = \beta_{suspend}^{cpu}, \tag{3}$$

where $\Delta t$ is the time difference between two successive events, and $n$ is the number of cores, which is two in this paper. $u_i^{cpu}$ and $r_i$ are the utilization and the normalized co-utilization rate of the $i$-th core during $\Delta t$. We figure out the utilization of each core from the usage statistics variable called *cpu_usage_stat* that cumulates CPU execution and idle time in the Linux kernel. $\beta_{single}^{cpu}$, $\beta_{dual}^{cpu}$, $\beta_{idle}^{cpu}$, and $\beta_{suspend}^{cpu}$ denote power coefficients (with the unit of $W$) for the single-core, dual-core, idle, and suspend states, respectively, at $freq$ MHz operating frequency. We obtain values of all power coefficients through a characterization process that we will describe later. The CPU power model can be improved to account for the thermal effect caused by elevated temperature since the temperature sensors in the smartphone are all accessible from the Linux kernel. However, it is beyond the scope of delivering the concept of this paper.

*2) Wi-Fi:* As we explained in Section III-A, we monitor the transmission events in the NIC device driver for the Wi-Fi module. We obtain the accurate event timestamp because the events happen immediately before and after the actual transmission occurs. In addition, the Wi-Fi module has a power saving mode. This mode allows the Wi-Fi module turns off its transmitter and receiver if it does not have any packet to transmit. In general, the power consumption of the Wi-Fi module depends on its power mode and the packet transmission rate [10]. Thus, we monitor the clock on/off event of the module and model the Wi-Fi module as follows:

$$P_{WIFI} = \begin{cases} \alpha^{wifi}\left(\beta^{wifi}_{ht} + u^{wifi} \cdot \beta^{wifi}_{weight}\right) & u^{wifi} > u^{wifi}_{threshold}, \\ \alpha^{wifi} \cdot \beta^{wifi}_{lt} & u^{wifi} <= u^{wifi}_{threshold}, \end{cases} \quad (4)$$

where $u^{wifi}$ is the packet transmission rate during the past 500 ms. $\alpha^{wifi}$ is 0 if the clock of the Wi-Fi module is disabled, and 1 otherwise. $\beta^{wifi}_{ht}$ and $\beta^{wifi}_{lt}$ denote power coefficients for the high and low packet transmission rates, respectively, and $\beta^{wifi}_{weight}$ is the weight factor of the Wi-Fi module based on the packet transmission rate. $u^{wifi}_{threshold}$ is the threshold rate of the packet transmission, and we pick the optimal threshold from our experiments.

*3) Cellular:* The cellular module has a similar issue to the Wi-Fi module because Android uses the cellular module as a network adaptor if the Wi-Fi connection is unavailable. We monitor the cellular events in the transmission routines of the cellular module device driver. The cellular module repeatedly changes its power mode between an awake and a sleep mode depending on its load, and it goes into the suspend mode when the user explicitly disables the cellular connection. The power consumption of the cellular module is related with its power mode and packet transmission rate [10]. Therefore, we monitor the clock on/off events of the cellular module as well as the packet transmission rate. Our cellular power model is given as

$$P_{CELL} = \begin{cases} \beta^{cell}_{awake} + u^{cell} \cdot \beta^{cell}_{weight} & \text{if power mode is awake,} \\ \alpha^{cell} \cdot \beta^{cell}_{sleep} & \text{otherwise,} \end{cases} \quad (5)$$

where $\beta^{cell}_{awake}$ and $\beta^{cell}_{sleep}$ denote power coefficients for the awake mode and the sleep mode, respectively, and $\beta^{cell}_{weight}$ is the weight factor of the cellular module based on the packet transmission rate. $u^{cell}$ is the packet transmission rate during the past 500 ms, and $\alpha^{cell}$ is 0 if the cellular module is in the suspend mode, and 1 otherwise.

*4) GPS:* The GPS power consumption depends on its power mode. The Android OS enables the GPS module when an application requests the current location of the smartphone and disables the GPS immediately after the location service is done. We monitor the GPS clock on/off event in the driver layer, and the power model is defined as

$$P_{GPS} = \alpha^{gps} \cdot \beta^{gps}_{on}, \quad (6)$$

where $\alpha^{gps}$ is 0 if the GPS clock is disabled, and 1 otherwise. $\beta^{gps}_{on}$ is the power coefficient for the GPS module.

*5) Display-related Modules:* The power consumption of an OLED display depends on the pixel color that the display shows [11], [12], so the conventional power estimation methods that only use a screen brightness for a liquid-crystal display (LCD) are not appropriate for the OLED display. However, previous researches [2], [4], [6], [8] only present the LCD power model, so they significantly underestimate the display power consumption when their power models are used for the state-of-the-art smartphones that have an OLED display.

In this paper, we present a contents-aware low-overhead OLED display model. Android draws its video frame to the display device through the framebuffer of the Linux kernel, which is a memory area containing the pixel data of a current video frame that the display shows. For taking into account of the OLED display, we read the pixel data from the framebuffer and calculate the average pixel intensity.

In addition, we find that there are correlations in terms of the utilization among several display-related hardware components. We analyze the interaction among the hardwares using the correlation coefficient calculated from the utilization of each component. The analysis shows that the correlation coefficients among the OLED display, GPU, and DSP are higher than 0.75. This means these components are strongly correlated to each other. Therefore, we model these components all together using the artificial neural network (ANN) that is more robust to the correlation of the input variables. We characterize the OLED power consumption as a function of the average pixel intensity and the utilization of the GPU and DSP modules, so we monitor the GPU and DSP clock on/off events in each device driver. The OLED display power model is given as

$$P_{DISP} = \sum_{k=1}^{p}\left[\sigma\left(\sum_{j=1}^{n} u^{disp}_{j}\beta^{disp}_{iw[j][k]} + \beta^{disp}_{ib[k]}\right)\beta^{disp}_{hw[k]}\right] + \beta^{disp}_{ob}, \quad (7)$$

where $p$ and $n$ are the number of nodes in the hidden layer and the input layer of the ANN model, and we use $p = 5$ and $n = 3$ in this paper. $\sigma$ is the sigmoid function that is given by $\sigma(x) = (1 + e^{-x})^{-1}$. $u^{disp}_1$, $u^{disp}_2$, and $u^{disp}_3$ are the average of sampled pixels intensity, DSP utilization, and GPU utilization, respectively. $\beta^{disp}_{iw}$, $\beta^{disp}_{hw}$, $\beta^{disp}_{ib}$, and $\beta^{disp}_{ob}$ denote power coefficients for the input node weights, hidden node weights, input node biases, and output node bias of the ANN model, respectively.

## C. Time Granularity

As we described in Section II, the existing sampling-based techniques cause excessive energy overhead depending on its sampling frequency. According to our experiment, the energy overhead of the sampling-based method gradually increases from 1 Hz sampling frequency and saturates at about 10 Hz. On the other hand, the event-driven method does not suffer from the overhead thanks to its asynchronous nature, so it can perform fine-grained estimation whose granularity is higher than the sampling-based methods. However, the existing event-driven technique [8] still make the power estimation in a low time granularity, i.e., it gives one estimation result per second no matter how many events are logged during this second. Thus, we refer its power metering granularity as 1 Hz, and our experimental results show that such coarse-grained power metering results in significant power estimation error. Therefore, we employ the fine-grained approach with 1 kHz power metering granularity because it is enough to cover most of the power spectrum in the smartphone load currents, as shown in Fig. 2. We observe that the granularity levels higher than 1 kHz considerably increase the overhead and do not provide much more gain though.

## IV. Implementation

## A. Parameter Extraction

The proposed method takes the non-disruptive event-driven approach using a debugging tool called KProbes [13], which

TABLE I
COEFFICIENTS OF THE FEPMA POWER MODEL. THE COEFFICIENT INDEX OF THE PROCESSOR DENOTES THE OPERATING FREQUENCY, AND THE COEFFICIENT INDEX OF DISPLAY-RELATED UNITS DENOTES THE NODE NUMBER AND INDEX OF COEFFICIENTS IN THE ANN MODEL.

| Component | Coeff. | Coeff. index | | Value (W) | Coeff. | Coeff. index | Value (W) |
|---|---|---|---|---|---|---|---|
| CPU | $\beta^{cpu}_{single}$ | 350 | | 0.95 | $\beta^{cpu}_{dual}$ | 350 | 1.08 |
| | | 700 | | 1.26 | | 700 | 1.53 |
| | | 920 | | 1.58 | | 920 | 1.98 |
| | | 1200 | | 1.89 | | 1200 | 2.61 |
| | $\beta^{cpu}_{idle}$ | 350 | | 0.86 | $\beta^{cpu}_{suspend}$ | N/A | 0.72 |
| | | 700 | | 1.04 | | | |
| | | 920 | | 1.35 | | | |
| | | 1200 | | 1.58 | | | |
| Display | $\beta^{disp}_{iw}$ | 1 | 1 | 2.24 | $\beta^{disp}_{hw}$ | 1 | -0.30 |
| | | | 2 | -1.33 | | 2 | 1.07 |
| | | | 3 | 0.83 | | 3 | 0.16 |
| | | | 4 | -0.88 | | 4 | -1.04 |
| | | | 5 | 2.45 | | 5 | -0.28 |
| | | 2 | 1 | -0.81 | $\beta^{disp}_{ib}$ | 1 | 2.94 |
| | | | 2 | -3.03 | | 2 | -9.63 |
| | | | 3 | -9.21 | | 3 | 19.91 |
| | | | 4 | -3.62 | | 4 | -2.68 |
| | | | 5 | 16.21 | | 5 | -12.62 |
| | | 3 | 1 | 2.65 | $\beta^{disp}_{ob}$ | N/A | 0.10 |
| | | | 2 | 9.51 | | | |
| | | | 3 | -1.99 | | | |
| | | | 4 | 2.34 | | | |
| | | | 5 | -4.72 | | | |
| Cellular | $\beta^{cell}_{awake}$ | N/A | | 0.90 | $\beta^{cell}_{weight}$ | N/A | 3.24E-4 |
| | $\beta^{cell}_{sleep}$ | N/A | | 0.54 | | | |
| Wi-Fi | $\beta^{wifi}_{ht}$ | N/A | | 0.68 | $\beta^{wifi}_{weight}$ | N/A | 2.21E-4 |
| | $\beta^{wifi}_{lt}$ | N/A | | 0.32 | $u^{wifi}_{threshold}$ | N/A | 850 |
| GPS | $\beta^{gps}_{on}$ | N/A | | 0.14 | | | |

is developed by IBM. KProbes gives a non-destructive way to set a breakpoint to an arbitrary function in the kernel and specify a callback handler function that is called whenever the breakpoint hits. In this paper, we use Samsung Galaxy Nexus as a target platform and implement an event profiler as a kernel module to set breakpoints to several device driver routines, and we define customized callback handler functions. The customized callback handlers monitor events of interest related to the system activities and power modes. In addition, it records the timestamp of each event as well as other useful information (e.g., function parameters and global variables).

It is important to obtain the per component power consumption to extract power coefficients for each component. However, as state-of-the-art commercial smartphones do not provide facilities to let us directly measure power consumptions per each component, we carefully design some usecases to disable and enable one component at a time while others are remained unchanged. We compare the total power consumption before and after we run these usecases to obtain the power consumption for each component.

We first characterize the multi-core processor. Our event profiler monitors a change of the operating frequency, utilization, and suspend state on the *cpufreq_notify_transition* and *omap4_cpu_suspend* functions in the Linux kernel. We implement a custom test-bench application to model the dual-core processor in the target platform using various usecases. The processor power consumption is measured while the test-bench is executing, and all the other hardware components are disabled or utilized at a constant usage level if they cannot be disabled. Meanwhile, the proposed event profiler logs the processor-related events. We characterize other components in a similar procedure. For each component, we run custom usecases that only make changes to that component and measure the system power consumption, and then we subtract the estimated processor power from the measured system power consumption so that the per component power consumptions are mutually exclusive to each other. Our event profiler monitors the power mode change and the packet transmission start/stop events in the *sdioh_request_packet* and *dhdsdio_sdclk* functions for the Wi-Fi module and the *hsi_ioctl*, *hsi_read*, and *hsi_write* functions for the cellular module. It also monitors the clock on/off events in the *omap2_clk_enable* and *omap2_clk_disable* functions of the OMAP device driver for the GPS, GPU, and DSP modules. The model parameters are extracted by the regression analysis based on the event logs and measured per-component power consumptions.

## B. Online Power Metering

After we characterize the devices and extract the necessary power coefficients, we input these information to our online power metering framework named FEPMA. For online usage, our event profiler, which is a kernel module, captures the system events and records necessary information such as operating frequencies and power modes. Our power meter application, which is executed in user-space, reads the event log and predicts the system power consumption based on the presented power models. In addition, the event profiler also gather a process identifier when it captures events. The process identifier shows which process raised the event, and it is used to trace back applications and accumulate the estimated power consumption of each application in the system. Therefore, the power meter can show the energy profiles for each application as well as the total system power consumption.

In order to estimate the OLED display power, the power meter reads the screen pixel data from the framebuffer of the Linux kernel and calculates the average pixel intensity. It repeatedly performs this process because changing the pixels in the framebuffer does not raise any event that the proposed event profiler can detect. The power meter reads the framebuffer twice per second and divides each video frame into 3-by-3 pixel blocks, and then it uses the center pixel of each block to calculate the average pixel intensity. Moreover, Android provides a feature to adjust the screen brightness even if the smartphone has an OLED display, which does not have a backlight module. In general, a controller of the OLED display adjusts its screen brightness using gamma correction depending on the brightness level that the user sets. Accordingly, the pixel data in the framebuffer remains unchanged even though the brightness level changes, but the brightness change of the display affects the power consumption of the OLED display. Therefore, we weight the average pixel intensity by the current screen brightness to compensate for the difference between the calculated average pixel intensity from the framebuffer and the actual average intensity. We use the weighted average pixel intensity as an input variable of the ANN display power model in Equation 7.

## V. EXPERIMENTS

### A. Power Metering Results

We use the NI-9227 DAQ from National Instruments to measure the load current of the target platform and also use the E3648A DC power supply from Agilent to provide a constant voltage. We measure the power consumption at a 50 kHz sampling frequency from the NI DAQ and use it as the golden reference. We also implement the state-of-the-art event-driven power estimation method presented in [8] as the baseline. The power coefficients for our target platform are characterized as shown in Table I. We evaluate the FEPMA method using several famous Android applications. First, we estimate the power consumptions using the FEPMA and baseline methods, respectively, while we execute the Android applications, and

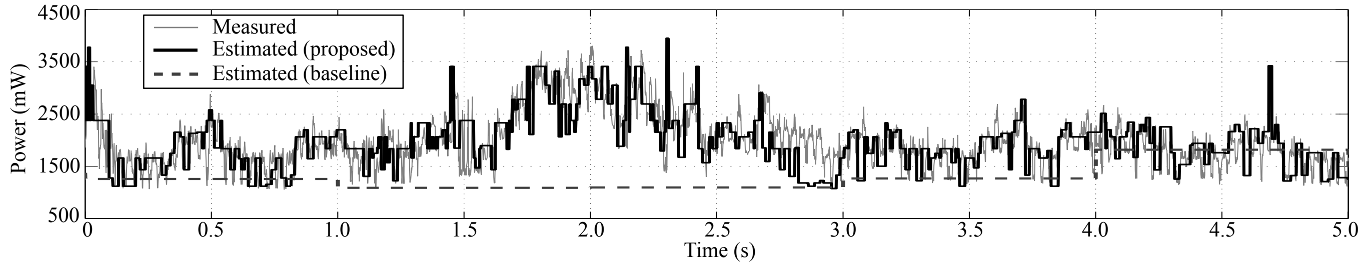

Fig. 3. Long-term measured power by the DAQ and estimated power by FEPMA while playing a music video using the Android movie player.

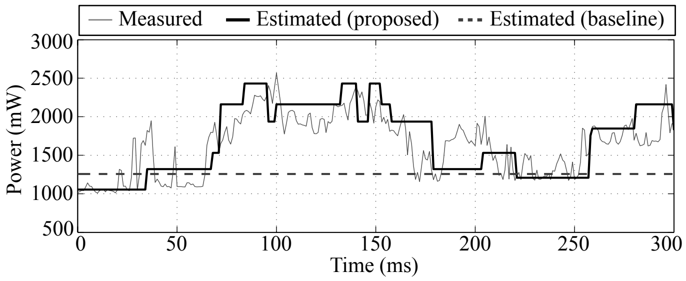

Fig. 4. Short-term power measurements by the DAQ and estimated power by FEPMA for video playback using the Android movie player.

then we compare our estimation result with the baseline result and the golden reference. Each set of comparison lasts for five seconds. This five-second time interval is the long term for the instantaneous power estimation, and it is also enough to analyze the state transition of hardware components and decide a proper DTM strategy for the current condition.

Figs. 3 and 4 show the long-term and short-term power metering results for video playback using FEPMA, the baseline, and the golden reference, respectively. Compared to the coarse-grained baseline results, our method yields more fine-grained accurate estimation results. The baseline method fails to capture the instantaneous power changes of hardware components because its estimation granularity is too low, and it significantly underestimates the power consumption of some components such as the multi-core processor, OLED display, GPU, and DSP due to the lack of consideration of these components and the unobservable device problem. In comparison with the golden reference, the root-mean-square (RMS) errors of the estimation results by FEPMA and the baseline in Fig. 4 are 293 mW and 538 mW, respectively. Table II provides the long-term evaluation results for various applications. The proposed method reduces the power estimation RMS error by up to 53.1% compared with the baseline.

## B. Overhead Analysis

We analyze the overhead of the proposed method in terms of computation and energy by measuring and comparing the processor execution time and the system power consumption of the target platform with and without our event profiler under the idle, normal load, and heavy load conditions, respectively. In this experiment, the analysis shows that the computation and energy overheads of the proposed method are 3.1% and 1.5% (22 mW on average), respectively. On the other hand, the baseline method shows 5.9% computation overhead and 34.9 mW energy overhead on average in the same conditions.

## VI. Conclusion

We introduce a novel sensor-less, event-driven power analysis framework called FEPMA for providing highly accurate and nearly instantaneous estimates of power dissipation in an Android smartphone. We monitor system events in the device driver layer of the Linux kernel to obtain necessary information such as power modes and activities of hardware components. By collecting the data at the lowest layer of the kernel, we manage to obtain accurate information and get access to many devices, which are not accessible in upper layer of the kernel. We present power models for identified components and perform a characterization process to extract the power coefficients for the proposed power models. The online power meter monitors system events and calculates the fine-grained power consumption of the smartphone based on the proposed power models. Experimental results show significant accuracy improvement compared to previous power estimation approaches. The estimation error of the proposed method is reduced by up to 53.1% compared with the baseline.

TABLE II
RMS errors of estimated power consumption by the baseline and FEPMA compared with the golden reference.

| Application | Baseline (mW) | FEPMA (mW) | Error reduction |
|---|---|---|---|
| Movie Player | 927 | 435 | 53.1% |
| Angry Birds | 1169 | 584 | 50.0% |
| Chrome Browser | 1324 | 693 | 47.7% |
| Skype (Wi-Fi) | 949 | 510 | 46.3% |
| YouTube | 987 | 564 | 42.9% |
| Skype (Cellular) | 1017 | 612 | 39.8% |
| Google Maps | 957 | 593 | 38.0% |

## References

[1] B.-G. Chun and P. Maniatis, “Augmented smartphone applications through clone cloud execution.,” in *HotOS*, vol. 9, pp. 8–11, 2009.

[2] A. Shye, B. Scholbrock, and G. Memik, “Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures,” in *MICRO*, pp. 168–178, 2009.

[3] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich, “Energy management in mobile devices with the cinder operating system,” in *EuroSys*, pp. 139–152, ACM, 2011.

[4] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, “Accurate online power estimation and automatic battery behavior based power model generation for smartphones,” in *CODES+ISSS*, pp. 105–114, 2010.

[5] M. Dong and L. Zhong, “Self-constructive high-rate system energy modeling for battery-powered mobile systems,” in *MobiSys*, pp. 335–348, 2011.

[6] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha, “Devscope: a nonintrusive and online power analysis tool for smartphone hardware components,” in *CODES+ISSS*, pp. 353–362, 2012.

[7] A. Pathak, Y. C. Hu, and M. Zhang, “Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof,” in *EuroSys*, pp. 29–42, 2012.

[8] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, “Appscope: Application energy metering framework for android smartphone using kernel activity monitoring,” in *USENIX ATC*, 2012.

[9] A. Carroll and G. Heiser, “An analysis of power consumption in a smartphone,” in *USENIX*, pp. 21–21, 2010.

[10] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, “Energy consumption in mobile phones: a measurement study and implications for network applications,” in *IMC*, pp. 280–293, 2009.

[11] D. Shin, Y. Kim, N. Chang, and M. Pedram, “Dynamic voltage scaling of oled displays,” in *DAC*, pp. 53–58, 2011.

[12] M. Dong, Y.-S. K. Choi, and L. Zhong, “Power modeling of graphical user interfaces on oled displays,” in *DAC*, pp. 652–657, 2009.

[13] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, “Probing the guts of kprobes,” in *Linux Symposium*, vol. 6, 2006.