Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices

MOHAMMAD ASHRAFUL HOQUE, University of Helsinki
MATTI SIEKKINEN, KASHIF NIZAM KHAN, and YU XIAO, Aalto University
SASU TARKOMA, University of Helsinki

Software energy profilers are the tools to measure the energy consumption of mobile devices, applications running on those devices, and various hardware components. They adopt different modeling and measurement techniques. In this article, we aim to review a wide range of such energy profilers for mobile devices. First, we introduce the terminologies and describe the power modeling and measurement methodologies applied in model-based energy profiling. Next, we classify the profilers according to their implementation and deployment strategies, and compare the profiling capabilities and performance between different types. Finally, we point out their limitations and the corresponding challenges.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement Techniques, Design Studies, Performance Attribute

General Terms: Design, Experiment, Measurement, Performance

Additional Key Words and Phrases: Energy profiler, power model, power consumption, energy debugging, battery model, fuel gauge, discharge curve, energy bug, energy hog, energy hotspot

ACM Reference Format:
Mohammad Ashraful Hoque, Matti Siekkinen, Kashif Nizam Khan, Yu Xiao, and Sasu Tarkoma. 2015. Modeling, profiling, and debugging the energy consumption of mobile devices. ACM Comput. Surv. 48, 3, Article 39 (December 2015), 40 pages.
DOI: http://dx.doi.org/10.1145/2840723

1. INTRODUCTION

Smartphones are powered up with batteries having limited capacity. Many of the hardware components found inside modern smartphones, namely displays, cameras, radios, and processors, draw considerable amounts of power. Meanwhile, smartphone applications today are getting more and more resource hungry. Obviously, in addition to the energy efficiency of the hardware itself, what matters to the battery life is how the hardware is used by the applications.

Many kinds of energy management mechanisms have been developed to reduce energy consumption [Vallina-Rodriguez and Crowcroft 2013]. These mechanisms are most often implemented on the operating system (OS) level and designed to operate without deteriorating or penalizing the application performance or user experience,

This research was supported by the Academy of Finland with grants 277498, 278207, and 268096, and the Helsinki Institute of Physics, Green Big Data project.
Authors' addresses: M. A. Hoque and S. Tarkoma, Computer Science Department, University of Helsinki; emails: {mohammad.hoque, sasu.tarkoma}@cs.helsinki.fi; M. Siekkinen, K. N. Khan and Y. Xiao, Computer Science Department, Aalto University; emails: {matti.siekkinen, kashif.khan, yu.xiao}@aalto.fi.

DOI: http://dx.doi.org/10.1145/2840723

hence remaining invisible to application developers. Examples include dynamic voltage and frequency scaling (DVFS) of microprocessors and power-saving modes of radios.

Previous studies have shown that in many cases the energy efficiency of a particular application can be dramatically improved through changes in the program code even in the presence of such management mechanisms. One reason is that these mechanisms are application agnostic and therefore are usually unable to optimize their way of working to match the application workload, even if application developers have done a good job. Streaming traffic scheduling represents a prime example of this kind of optimization where the time spent by the radio powered on is minimized through clever timing of data transmissions and reception [Hoque et al. 2014b]. Another reason is that application developers may simply implement a particular task in an energy-inefficient manner, which may not cause any other visible symptoms than rapidly draining the battery. Such a suboptimal piece of code is sometimes called an *energy bug* [Pathak et al. 2011].

All of the preceding means that it is crucial for developers to be aware of the energy consumption behavior of smartphones when conducting particular operations. Without that ability, it is difficult for developers to optimize the program code for energy efficiency. This article focuses on software-based solutions for analyzing and estimating smartphone energy consumption. A basic functionality of these software is to provide information about battery usage, such as the energy being consumed by the whole device, by each hardware component such as CPU and display, and/or by each application/service running on the device. We call this functionality energy profiling and the software with such functionality *energy profilers*.

Smartphone energy profiling is a multifaceted problem. The first issue is that most smartphones do not by default even report the total amount of current the device draws at a particular moment of time. An option is to measure it using external instruments, such as Monsoon Power Monitor [Monsoon 2014], BattOr [Schulman et al. 2011], and NEAT [Brouwers et al. 2014]. However, this method requires opening the phone and attaching the measurement unit to the phone's physical battery interface, which is rarely desirable and sometimes very difficult, as is the case with Apple iPhones, whose batteries are not easily accessible. Concerning the potential decrease in output voltage with the remained battery capacity, developers may choose to replace the battery with an external power supply during measurement, which can provide more accurate measurement but is not portable. To this end, several methods have been proposed to estimate the instantaneous power draw from the smartphone's battery API, which typically is able to report the voltage and the state of charge (SOC) at certain intervals. On a limited number of smartphone models, the battery API is able to report current, which mitigates this part of the profiling problem.

The second challenge in energy profiling is to understand how the total energy consumption is distributed among the different hardware components. The previously mentioned power measurement methods can tell the total power draw of the device but cannot tell directly the major underlying power sinks (i.e., the hardware resources being utilized by the application and their contribution in total power consumption). Although it is possible to extract the contributions of some individual subcomponents through extensive analysis of the measurement data, the effort requires domain specific knowledge, and it is not always feasible to do so. Overcoming this challenge requires power modeling in most cases. A power model represents the power draw of a system as a function of subsystem specific variables. These variables are chosen in a way that their values can be continuously monitored by software, which enables continuous assessment of the power drawn by the system and subsystems without the need for external instruments. The research community has taken many different approaches to smartphone power modeling, which we will review in detail in the following sections of the article.

The final challenge in smartphone energy profiling is to enable attributing the system and subsystem energy consumption to different pieces of application program code. Indeed, an energy profiler that is able to point out the most critical parts of program code from the energy consumption perspective is clearly more useful to software developers than a profiler that can only provide an energy consumption profile per hardware component. Some of the profilers that we survey in the rest of the article have this capability. In general, this feature requires the capability of tracing the execution of applications on a fairly fine granularity and being able to match the instantaneous power draw to specific instances of program code execution.

Our approach in this survey is to take a broad look into different kinds of software-based energy profilers for smartphones. It is broad in the sense that we survey the solutions from the most basic ones that are able to just report total instantaneous system power to the richest kind that are able to provide energy consumption profiling on the level of application program code.

Besides the actual energy profilers, there are several software tools that try to detect abnormal energy usage by different applications, subcomponents, and the reasons behind such behavior. We define them as *energy diagnosis engines*. They also actively apply or suggest users the prognosis for the energy-buggy applications. However, such a system may depend on an energy profiler and run on the mobile device or in a separate system. We discuss theses tools separately in Section 9.

This survey is composed of five major parts. First, we familiarize the reader with the terminologies involved in software-based energy profiling work and used throughout this survey in Section 2. Second, power measurement is an integral part of the power modeling, and we present the power measurement methodologies in Section 3. Third, power modeling is the heart of the software-based profilers, and we explain different power modeling methodologies in Section 4. We next compare the existing power profilers in Sections 6, 7, 8, and 9 according a taxonomy presented in Section 5. Finally, we address the limitations and challenges with existing profilers from the accuracy and usability perspectives in Sections 10 and 11, which also serve as our suggestions on how to advance the state of the art of software-based energy profilers for smartphones.

2. MODEL-BASED ENERGY PROFILING

To profile the energy consumption of the system, its subcomponents, or specific applications, power models are needed. A model-based energy profiler provides some kind of energy consumption profile of the mobile device, one of its subsystems, or a piece of software running on it. Power measurement is an integral part of building a power model and can be implemented in alternative ways. In this section, we describe the anatomy of an energy profiler and the related activities, including power modeling and measurements.

2.1. Basic Terminology

The literature is rich with different kinds of energy profiling solutions. These solutions are usually presented as unique tools, although they are in fact the combinations of different kinds of underlying techniques. To provide deeper insight into these solutions, it is essential to identify the key components of each energy profiler and to analyze the relevance between them. In the literature, the terminology has not been used consistently. For example, in some cases, the line separating power measurements and model-based profiling is thin, although a particular approach to estimating power consumption could be described using either of the two terms. Hence, it is important that we clearly define the terminology we use:

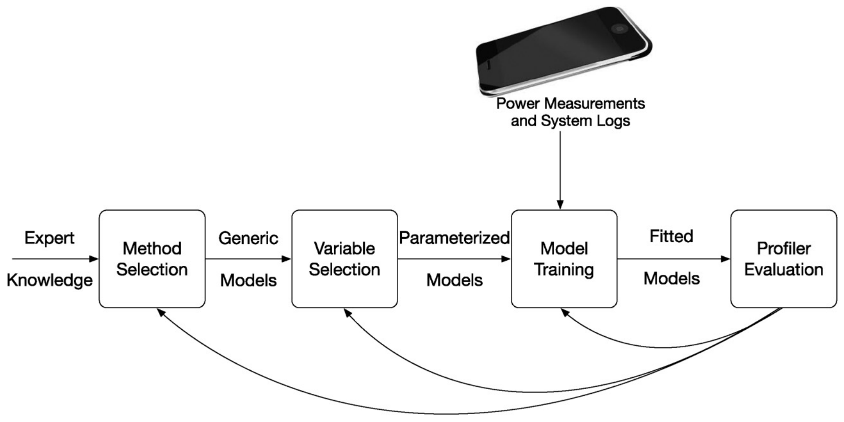

Fig. 1. The process of energy or power profiling includes modeling, estimation, and a feedback loop through validation and verification to refine and recalibrate the models.

—*Power measurement* is the act of obtaining power (or current) consumption values using specific hardware. It involves no models implemented in the software and calibration. The most common example is power measurement using an external instrument, such as Monsoon Power Monitor [Monsoon 2014], connected to the battery interface of a smartphone. Another example is direct measurement of current from the smart battery interface of a smartphone implemented with special-purpose embedded electronics. The Nokia Energy Profiler (NEP) [Creus and Kuulusa 2007], for instance, relies on this approach to obtain the current draws.

—*Power model* is a mathematical representation of power draw. It is a function of variables that quantifies the impacting factors of power consumption, such as the utilization of a hardware subcomponent and the number of packets delivered through a wireless network interface, with the desired power draw as output. Usually the values of these variables can be directly obtained from measurement carried out on the smartphone or in the network. A power model can characterize a single subsystem, a combination of them, or even a whole smartphone (system-level model). A simple example of a subsystem power model is a coarse-grained power model of display that is a function of a single variable: brightness level.

—*Power estimation* reports power draw of a smartphone or its subsystem based on power model(s). The accuracy of the power estimates depends on the accuracy of the power model in use.

—*Power/energy profiler* is a system that characterizes power and/or energy consumption of a smartphone. We distinguish profiling from power measurement by specifying that a profiler relies on power models by definition. Hence, a profiler provides power or energy estimates as opposed to power measurements that can be obtained with a power monitor. Furthermore, different profilers work on different abstraction levels, such as system, application, process, or task, whereas power measurement only provides power consumption of the hardware under measurement, most often the entire smartphone.

2.2. Constructing an Energy Profiler

Figure 1 illustrates the process of constructing a model-based energy profiler. The figure divides the process into four phases. The process starts with the expert selecting the modeling method using her domain-specific knowledge. After this, the actual power modeling phase follows, in which variables are selected and the models are trained using power measurements and system logs. The system logs refer to the actual variables used in the power models and the training is in essence computing the coefficients of the model variables. Statistical learning techniques are commonly applied in these phases. In Section 4, we will examine the different power modeling approaches that are relevant to these two phases.

The power models are then combined into the actual profiler that monitors the model variables and provides power consumption estimates. The profiler is evaluated with the help of additional power measurements that are compared to the power estimates to characterize their accuracy. This phase provides valuable input for the expert overseeing the model generation and usage process. According to the input, the choice of modeling approach and/or variable selection can be reevaluated, whereas the model can be retrained with a more comprehensive training dataset.

3. OBTAINING SYSTEM-LEVEL POWER CONSUMPTION

We first discuss two alternative ways to measure the overall power consumption of a smartphone: using external instruments and by means of self-metering. The collected measurements are parts of the input for training power models of the smartphone.

3.1. Power Measurement Using External Instruments

External instruments, such as the Monsoon Power Monitor, can be used for directly measuring the current drawn by the smartphone. Another way to use an external instrument is to connect a voltage meter across a resistor that is connected in series with the power supply of the smartphone, which allows computing the current from the change of the measured voltage. It is also possible to perform these measurements with the original battery as the power supply or using an external power supply. In the former case, the battery effects, such as impact of the SOC, are included in the measurement. The external power monitors are the gold standard for mobile device power analysis due to their high precision and accuracy. They are limited by requiring the laboratory settings and are therefore not feasible for large-scale deployment.

3.2. Self-Metering

The second approach is called *self-metering*, which means that the smartphone is equipped with sufficient capabilities to infer system-level power consumption without the help of external instruments.

Battery models, voltage, and state of charge. Let us first briefly introduce three measurement metrics related to battery characteristics before we take a closer look at the different self-metering approaches. The metrics include the terminal voltage, open circuit voltage (OCV), and SOC. The terminal voltage is the measurable voltage of the battery across its terminals, whereas OCV defines the battery voltage without load. The terminal voltage (V_t) drops and hence differs from the OCV (V_{OCV}), when current is drawn from the battery, due to its internal ohmic resistance, R, as follows:

$$V_t = V_{OCV} - I \cdot R. \tag{1}$$

The preceding is also known as the Rint battery model, and the equivalent circuit is drawn in Figure 2. Compared to that model, the Thevenin model introduces a parallel RC network in series on top of the Rint model as shown in Figure 2. Accordingly, the Thevenin model consists of OCV, internal resistances, and the equivalent capacitance. The internal resistances include the ohomic resistance R and the polarization resistance r. The capacitance C_{Th} describes the transient response of the battery while charging and discharging. V_c is the voltage across C_{Th}. The Thevenin model can be

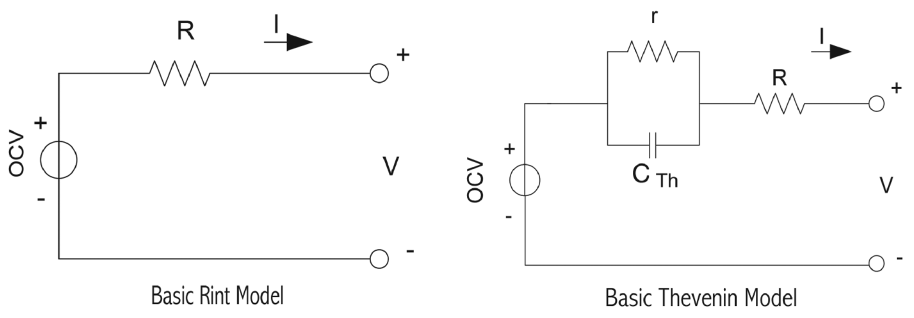

Fig. 2. OCV estimation models.

expressed as

$$V_t = V_{OCV} - V_c - I \cdot R. \tag{2}$$

The preceding models suggest that the exact OCV can hardly be measured on a powered-up smartphone; however, the measured terminal voltage comes close to it when all of the hardware components stay in low power mode.

The SOC defines the current charge status of the battery or the remaining battery capacity in a percentage. For example, 0% implies an empty battery and 100% implies a fully charged battery. The alternative way to express the remaining capacity is using state of discharge (SOD), in which 100% implies an empty battery and 0% implies a fully charged battery. However, being able to tell the SOC is useful for smartphone users so that they know when the device battery needs to be recharged. In the context of energy profiling, the SOC plays another important role, providing a means to estimate the average dissipation of current given a constant load. It is done so that the SOC is recorded before and after applying specific load to the smartphone. The change in the SOC, together with the knowledge of the battery capacity, directly yields the amount of energy consumed by the phone during the measurement interval from which the average current drawn by the device while it was under the specific load can be calculated, as the duration and the OCV are known.

State of charge estimation. Typically, the SOC cannot be measured directly. Instead, there are two approaches to estimating it [Rezvanizaniani et al. 2014]: (i) voltage-based method and (ii) Coulomb counting. The first method method uses either of the battery models discussed in the earlier section and converts the terminal voltage to SOC using OCV lookup tables. It is common to use a so-called battery discharge curve for expressing the relationship between the OCV and the remaining capacity as the battery discharges over time (see Figure 7 in Section 6.3). This discharge curve is always a strictly decreasing function. Given a particular value of the OCV, the SOC can be determined. The drawback is that the mapping varies with battery properties, such as model and age, which means that for accurate estimation of SOC, a personalized discharge curve must be generated for each device and be updated from time to time.

The Coulomb counting technique determines the SOC by accumulating the current drawn by the system over time. This method requires the ability to directly sense the current, because of which it is counterintuitive to discuss it here. The reason is that it is possible that a device that has the capability to sense current drawn from the battery only uses it to estimate the SOC and does not expose the instantaneous current to the applications through the OS. Therefore, an energy profiling application needs to rely on SOC-based power estimation in such a case. In this case, SOC can be calculated as

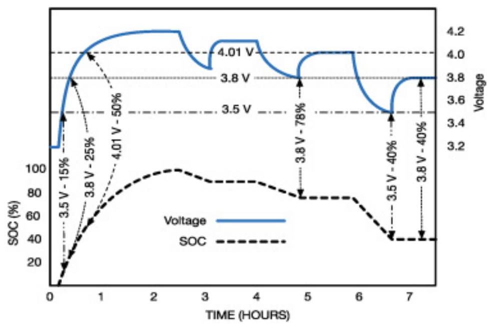

Fig. 3. Instantaneous voltage does not correlate with the SOC [Maxim 2014b].

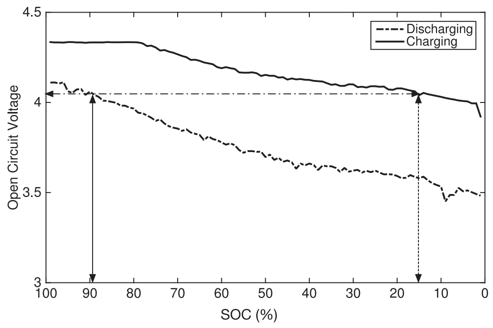

Fig. 4. OCVs of Samsung Galaxy S4 while charging and discharging do not correlate with SOC.

$$SOC = SOC_{init} - \int \frac{I_{bat}}{C_{usable}} dt, \tag{3}$$

where C_{usable} depends on the battery capacity reduction due to age, temperature, and charging cycles, and losses due to inactivity of the battery.

Since the SOC is estimated, terminal voltage, OCV-based, and Coulomb counting approaches may suffer from error, and the profilers relying on SOC also inherit the error.

In case of the voltage-based SOC estimation, the estimation error can be significant, as the battery voltage varies with load, temperature, and age. The Rint model for OCV does not consider the transient nature of lithium-ion batteries, and thus error can be significant under dynamic load of the system.

Due to the dynamic load experienced in a system, there remains the possibility of error even with a sophisticated voltage and load lookup table. For example, in Figure 3, we can see that battery voltage 3.81V occurs at 25%, 78%, and 40% of the SOC. Figure 4 shows a similar example for OCV-based SOC estimation. We can see that an OCV represents multiple SOCs while charging (15%) and discharging (90%), and consequently the SOC error will be significant unless separate OCV lookup tables are maintained for charging and discharging.

In the case of Coulomb counting, there two sources of error. First, there is an offset current accumulation error. The offset current results from the current sense and analog-to-digital conversion [Texas Instruments 1999]. This accumulation error increases over time and contributes to inaccuracy of the SOC, unless it is compensated with some voltage relaxation method such as keeping the device idle for a long time. Second, there is a usable capacity estimation error that is related to the age of the battery, temperature, and charging or discharging rates. The traditional approaches to estimate the usable capacity are the charging cycle and lookup tables with respect to the temperatures and rates [Hoque and Tarkoma 2015].

Fuel gauge and battery APIs. The SOC is estimated by a hardware component in a mobile device, called the *battery fuel gauge*. The voltage-based fuel gauges read battery voltage from the battery and are easy to implement. On the other hand, Coulomb counter–based fuel gauges are required to be instrumented with a sense resistor in the charge/discharge path. Under current flow, an analog-to-digital converter (ADC) reads the voltage across the sense resistor and then transfers the current value to a counter. The timing information is provided by a real-time counter to integrate current to Coulomb. The latest phones, such as Nexus 6/9, use Coulomb counter–based fuel gauges [Android 2014a].

From the perspective of an energy profiling application, the self-metered information is provided through the smartphone's battery API, which is a way for the OS to expose information about the battery status, such as the Android's BatteryManager. The exact information provided by the battery API depends on the device model and the fuel gauge type. In some cases, the API can directly provide information about the current draw, battery voltage, and temperature. The API updates this information periodically and whenever there is a change in the SOC depending on the fuel gauge. The update rates vary from 0.25Hz to 4Hz [Zhang et al. 2010]. Later we will see how the profilers that rely on the self-metering approach utilize voltage, current, or SOC from these updates.

4. POWER MODELING METHODOLOGY

In this section, we take a closer look at the different approaches for smartphone power modeling.

4.1. Types of Power Models

The approaches to modeling power consumption of a smartphone can be roughly divided into three categories based on the kind of input variables the model uses: utilization-based models [Zhang et al. 2010], event-based models [Pathak et al. 2011], and the ones based on code analysis.

1. Utilization-based models. Utilization-based models account for the energy usage of a subcomponent by correlating the power draw of that component with measured resource usage. For an application or process, its power model includes variables that reflect the resource consumption of all of the different subcomponents that are active while running that application.

A good example of utilization-based approach is the widely used method for modeling power consumption of the computing subsystem using hardware performance counters (HPCs). Such a method leverages the fact that modern microprocessors expose their internal activity through several event counters, such as instructions per cycle (IPCs), fetch counters, miss/hit counters, and stalls. The idea is that the amount of power required for executing a software is proportionate to the amount of activity that happens inside the microprocessor. Contreras and Martonosi [2005] relied on counters for modeling both CPU and memory power consumption. For modeling memory power consumption, the authors considered cache misses and stall counters. Li and John [2003] characterized the power behavior of a commercial OS for a wide range of applications using IPC. Singh et al. [2009] proposed a power model using counters for an AMD Phenom processor. Bircher and John [2007] explored the usage of counters to estimate the power consumed by subsystems outside the processor. An example of mobile device power modeling work based on HPCs is presented in Joule Watcher [Bellosa 2000].

2. Event-based models. Utilization-based power models are good at capturing linear relationships between resource and energy consumption of the hardware being modeled. However, some of the smartphone hardware components exhibit nonlinear energy consumption characteristics. This behavior is often characterized with a *tail energy* concept, which refers to the fact that a specific piece of hardware remains powered on for some time after it is no longer actively used. For example, wireless radios typically remain powered on for a particular timer-specified amount of time after the last bit has been transmitted or received [Balasubramanian et al. 2009]. For this reason, event-based modeling has been adopted. Events allow more accurate characterization

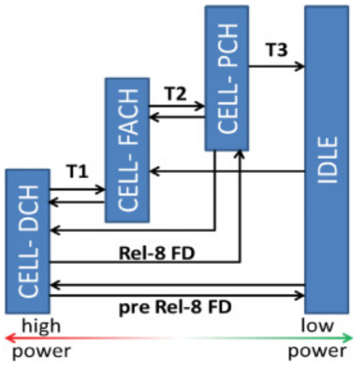

Fig. 5. HSPA cellular network steps involved in data connectivity.

of the power in certain cases where utilization-based approach does not perform well. Obviously, a mixture of the two approaches may also be used.

A representative example of an event-based modeling approach is one that builds the power model based on system calls [Pathak et al. 2011]. Tracking system calls provides a clear indication of which hardware components are used by a specific application or process and in which way. For example, the tasks related to I/O are exposed through `read` and `write` system calls.

The duration of the active power state of the corresponding hardware subcomponents depends on the volume of the I/O, which are specified in the parameters of these system calls. When the actual I/O tasks are finished, the tail energy can be estimated using `close`-like system calls. Pathak et al. [2011] considered the tail energy behavior of different hardware components.

As another example, the states of radio resource control (RRC) protocol in 3G cellular network communication and the transitions among the states are presented in Figure 5. The figure highlights that power consumption is highest at the CELL_DCH state. The amount of time required to be in each state depends on the length of the timer and the data activity at the moment. The timer lengths in fact depend on how the network equipment is configured by the operator. If fast dormancy (FD) is supported, the device will directly switch from the CELL_DCH to the CELL_PCH or the IDLE state when the FD timer expires. For additional details, we refer interested readers to the work presented by Siekkinen et al. [2013].

3. Models based on code analysis. The third category of models relies on the analysis of the program code to be executed. The advantage of this approach is that it can estimate energy consumption without even executing the software on a real system. This approach is used less frequently, as the energy consumption is often context dependent, which is difficult to account for without actually running the program code in a real device. For example, poor wireless link quality that prolongs the transmission time of a file and affects the energy consumption can be captured by a utility-based model that tracks the bits transmitted as a function of time but not by means of code analysis. An example of code analysis approach is an instruction-level model, which works by associating the power consumption of a piece of software with each instruction executed and requires the evaluation of power dissipation for each of the instructions of the software considered. A similar method can be applied for a function, procedure, or subroutine.

4.2. White Box Versus Black Box Modeling

The modeling approaches can be further distinguished by the amount of available a priori information about a hardware component being modeled. A purely black box approach, as the name suggests, has no a priori information about the hardware component's power consumption behavior, whereas in the case of white box modeling, the behavior is well understood.

White box power modeling typically captures the power consumption behavior of the hardware using finite state machines (FSMs) that describe the power states of the system and transitions between power states. This approach requires precise understanding of the power consumption behavior of the hardware. Specifically, the events that trigger a transition from one power state to another must be known and well understood. Training of the model is not required beyond simply measuring the absolute power draw of the hardware at different power states. This approach works well when modeling the power consumption of the wireless communication subsystems of a smartphone. The power draw of a radio can be abstracted accurately enough using a simple set of power states that correspond to the fraction of time the radio is fully powered on. Transitions between power states depend on the link layer protocols used, but they are usually triggered by inactivity timers and thresholds associated with the transmission data rate. Examples of such modeling are presented in Xiao et al. [2014] and Hoque et al. [2013b].

In contrast, black box modeling always requires model training. The main method is regression analysis and, in particular, linear regression. A model based on regression analysis captures the relationship between an input workload described with regression variables and the power consumption. Typically, some a priori knowledge is available, which helps, for example, to select the regression variables. Linear regression is a natural choice when constructing a utility-based power model where CPU usage, screen brightness, and network activity could be captured by regression variables [Xiao et al. 2010]. It is simple and efficient but also limited by the linearity assumption, but transformations sometimes can overcome this limitation. Examples of profilers that use linear regression are DevScope [Jung et al. 2012] and V-edge [Xu et al. 2013]. There are some profilers that do both white and black box modeling, such as those proposed by Banerjee et al. [2014].

5. TAXONOMY OF MODEL-BASED ENERGY PROFILERS

As explained in Sections 2 and 4, an energy profiler is a piece of software that measures and monitors the energy consumed by a subcomponent of a mobile device, the whole device, or an application. Table I summarizes the aspects of model-based power profilers that we discussed in previous sections. The power or energy can be modeled as a linear or nonlinear function of several variables. The computation of the variables can be conducted on a desktop computer or in the cloud, or even on the mobile device itself. The data, defining the variables in the model, may include hardware resource utilization statistics, a system call trace, and operating or power states of different hardware components. The modeling methodology can be white or black box, or a combination of the two. The power model may depend on the measured power values that can be measured from the smart battery interface or using physical power measurement tools such as a power meter.

Table II illustrates a classification of the existing power profilers. The two main axes along which we differentiate the profilers are whether the model construction and training happens on the smartphone or not and whether the profiler runs on the phone or not. As a result, we identify three main categories of profilers as follows:

Table I. Guiding Questions for Profiler Classification

Criteria	Choices
What is the level of profiling granularity?	Whole device (system), subcomponent, application, or API
Where to train power models?	On device (requires self-metering capability, cf. Section 3.2)
	Of device (e.g., on PC, in the cloud)
Where to get the model predictors?	Metrics that measure the utilization of hardware resources (e.g., HPCs)
	Traces of software execution (e.g., system call trace, traffic traces)
	Hardware/software operating modes
Which modeling methodology to use?	White box
	Black box (e.g., linear regression)
	Combination of white box and black box methods
How to get the power values for model training?	Use of physical power meters (e.g., Monsoon Power Monitor)
	Performing self-metering

Table II. Classification of the Energy Profilers

Profiler Name/ Author	Profiling Granularity	Model Type	Model Construction	Deployment Type
Nokia Energy Profiler	System level	Not applicable	On-device	On-device
Trepn Profiler	System, subcomponent, and application levels	Not applicable		
PowerBooter	System level	Linear regression		
Sesame	System and subcomponent levels	Linear regression, utilization		
DevScope	System and subcomponent levels	FSM, linear regression		
AppScope	Application level	DevScope models		
V-edge	System and subcomponent levels	Linear regression		
Android Power Profiler	System, subcomponent, and application levels	Utilization	Off-device	On-device
PowerTutor	System, subcomponent, and application levels	FSM		
PowerProf	API level	Genetic model		
PowerScope	Process and procedure levels	Code analysis	Off-device	Off-device
Joule Watcher	Thread level	Utilization (HPC)		
Eprof	System, subcomponent, and application levels	System call tracing and FSM		
Banerjee et al. [2014]	System, subcomponent, and application levels	System call tracing, linear regression		
Shye et al. [2009]	System and subcomponent levels	Utilization, linear regression		

—*On-device profilers with on-device model construction*: These profilers do not need offline tuning, measurement, or pretraining of power models. Rather, they generate power models at runtime based on the information gathered from the system, hence relying on self-metering. On-device models are important for two reasons. First, the hardware component of a mobile device may change. For example, a memory card is plugged into a phone or the device may change, and power consumption of the subcomponents may vary among the devices of the same model or different models.

Second, the usage of the system or applications may be different for different users, and users' behavior evolve with time [Falaki et al. 2010]. Therefore, on-device models enable not only avoiding complex device instrumentation but also enable device and usage agnostic energy profiling.

—*On-device profilers with off-device model construction*: These profilers also run on mobile devices but rely on power models that have been pretrained in laboratory settings. They use these models with the on-device information to characterize the energy consumption. For example, the power consumption of different subcomponents are modeled beforehand using an external power measurement tool, and later on the on-device usage statistics are used for estimating the energy consumption. Unlike the previous category, the models of these profilers are device specific, and thus their accuracy may vary significantly among different devices.

—*Off-device profilers*: These profilers estimate the energy consumption of applications or hardware components of mobile devices on a desktop machine or in the cloud rather than in a mobile device. Their profiling models are developed in the laboratory with the help of some external power monitoring tools such as the Monsoon Power Monitor or any simulation tool. Therefore, these profilers can often characterize the power consumption of the device, applications, and subcomponents more accurately or in a finer granularity, such as per method call instead of an entire application. In general, these kinds of profilers are mostly useful for application developers and system architects but not for regular users.

There is also a fourth category, namely off-device profilers with on-device model construction, but we exclude it since we are not aware of any work that focuses on these profilers. In most cases, if the model construction can be performed on-device, it makes sense to do the profiling on-device as well instead of transferring the input required by the model from the device to external profiler.

The profiling granularity of the different profilers varies. Granularity in this context refers to the capability of a profiler to dissect the total energy consumption of the smartphone. Subcomponent- and application-level profiling is more complex than system-level profiling simply because more detailed information is required about the underlying system behavior and application execution. For example, PowerTutor utilizes the component-level power models provided by the PowerBooter to estimate the application-specific energy consumption by attributing subcomponent-specific energy consumption to the application.

In the upcoming sections, we present the different energy profilers according to the three categories mentioned earlier. The description of the profilers also follow the chronological order as illustrated in Table II.

6. ON-DEVICE PROFILERS WITH ON-DEVICE MODEL CONSTRUCTION

The profilers of this category do not require the offline support for measurements or model calibration. They overcome these limitations by replacing the instrumental power measurements with self-metering. Examples include the NEP, Trepn Profiler, PowerBooter, Sesame, DevScope, AppScope, and V-edge. In this section, we first describe these energy profilers and then explore their similarities and differences.

6.1. Nokia Energy Profiler

The NEP is a stand-alone on-device power measurement software for Nokia's Symbian devices. It is one of the pioneers in the current trend of on-device power profiling. However, the NEP does not have practical relevance anymore, as the Symbian devices are no longer on the market. It displays the runtime energy consumption of the system in watts and amperes by monitoring system and networking activities.

Simultaneously, the NEP can monitor the network signal strength and the cellular network connectivity status of the mobile devices [Creus and Kuulusa 2007]. It also displays the voltage discharge curve. Most of the information is displayed as temporal graphs and can be exported as CSV files. The smart battery interface of Nokia Symbian devices provides both voltage and current sensor readings, and NEP's power measurement is implemented based on these two.

This tool is manually used by a user. Once activated, it starts profiling the system. The idea is that the user will run some testing applications and then visit the NEP interface to examine the energy usage of the target applications. The user can examine the total power consumption and the network usage of the applications as well. The NEP's sampling frequency has been limited to 4Hz to curb the resource consumption of the profiler itself.

6.2. Trepn Profiler

The Trepn Profiler [Qualcomm 2014b] is akin to the NEP for profiling the hardware usage and energy consumption. Trepn is developed by the Qualcomm community and works on devices with Snapdragon chipset-based Android devices. It is a user space application and can profile the usage and power consumption of the CPU, GPU, Wi-Fi, wakelocks, memory, SD card, and audio, as well as the runtime energy consumption of the whole device. Unlike NEP, it can provide fine-grained subcomponent-specific energy consumption. However, Trepn requires additional hardware instrumentation in the device, called the *Mobile Development Platform* (MDP). MDP is powered with Embedded Power Management SOC, which collects readings from the sense resistors and converts to current for individual hardware components [Qualcomm 2014a], such as the CPU, GPU, and Wi-Fi. Trepn also depends on a special fuel gauge chip with the integrated power management IC, which controls the distribution of power from the battery [Qualcomm 2014a]. For the usage statistics of different hardware components, Trepn depends on the `proc` and other system files. Although Trepn samples information after every 100ms, it can adapt the sampling rate to the system load to avoid the overhead of the sampling.

Similar to the NEP, it also offers different modes of information visualization. Trepn also provides an overlay view of different graphs and charts in the foreground so that the application developers can associate the performance of applications with the resource utilization and energy consumption at runtime. Meanwhile, it allows exporting the real-time raw data for offline analysis. Trepn further enables debugging the application performance by catching the Android intents and logging the application states along with other data points. Finally, Trepn can be controlled from an external application and thus facilitates automated profiling.

6.3. PowerBooter

PowerBooter [Zhang et al. 2010] automatically generates power consumption models by correlating the power consumption of individual subcomponents with the state of the battery with regression. Figure 6 illustrates the key steps involved in the model generation mechanism: obtaining battery discharge curves for individual hardware components, measuring power consumption of the components, and generating models.

The first step is to construct the discharge curves for each individual components. For that, PowerBooter uses the battery interface update. Figure 7 illustrates one example curve, which is a monotonically decreasing curve and expresses the relationship between the battery voltage and the SOD. The steepness of this curve depends on the discharge current, room temperature, and age of the battery. In addition, different batteries may produce different curves. Consequently, PowerBooter characterizes individual batteries by discharging a fully charged battery completely with constant

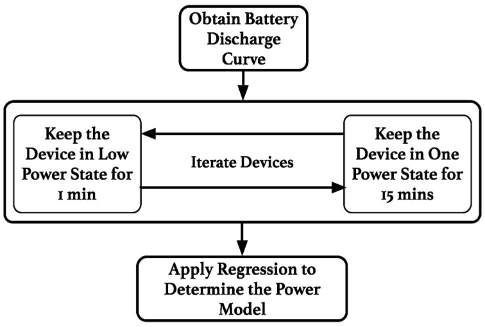
Obtain Battery Discharge Curve
Keep the Device in Low Power State for 1 min
Iterate Devices
Keep the Device in One Power State for 15 mins
Apply Regression to Determine the Power Model

Fig. 6. Overview of the PowerBooter model generation for multiple devices.

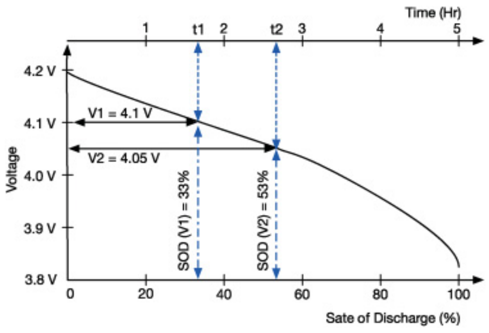

Fig. 7. Battery discharge curve: the status of battery voltage as discharge continues over time.

current and maintains a linear relationship between the SOD and the discharge time. Note that it applies piece-wise linear functions to represent the relationship between SOD and uses the battery output voltage.

The energy consumption measurements are carried in the second phase. In this phase, the states of an individual component are tuned while keeping the other components in lower power states or in some static configurations. For example, while determining the power consumption of CPU at different frequencies, the display, Wi-Fi, GPS, and cellular interface are disabled. The battery is discharged for 15 minutes, and the device is kept idle for 1 minute before and after the discharge interval. During the measurements, the interdependencies among different subcomponents are also considered, such as the interdependency between Wi-Fi and CPU power consumption, by monitoring the power states of the other components while exercising a particular component.

In the third phase, the power consumption of the components is derived from the measurements done in the second phase. The activities over this interval are mapped to the change in the SOD, which in turn is converted to the energy consumption. The energy consumption during an interval is calculated as $P \times (t_2 - t_1) = E \times (SOD(V_2) - SOD(V_1))$, where P is the average power consumption over time interval $[t_1, t_2]$, E is the related battery energy with respect to the battery capacity, and $SOD(V_1)$, $SOD(V_2)$ are the states of discharge at voltage V_1 and V_2, respectively, as illustrated in Figure 7. Finally, the regression is applied to generate power models.

6.4. Sesame

Sesame [Dong and Zhong 2011] also uses self-metering for generating the power model automatically. Differently from PowerBooter [Zhang et al. 2010], Sesame relies on getting instantaneous current readings directly without the need to resort to SOC-based estimation while generating the power models. This design decision limits the usage of Sesame to the phones that have OCV-based fuel gauge chips. The main novelty compared to PowerBooter is that Sesame uses statistical learning for generating power models that have higher accuracy and rate compared to the battery interface. Figure 8 illustrates Sesame's architecture. It has three subcomponents: a data collector, a model constructor, and a model generator.

In addition to the battery interface, the data collector collects data about the usage of different subcomponents from several sources, such as `proc` and `dev` system files. Sesame further considers the power status of different hardware components. However, different sources have different update rates, which may also depend on other activities. Therefore, reading from these sources at a higher rate or lower rate than the actual

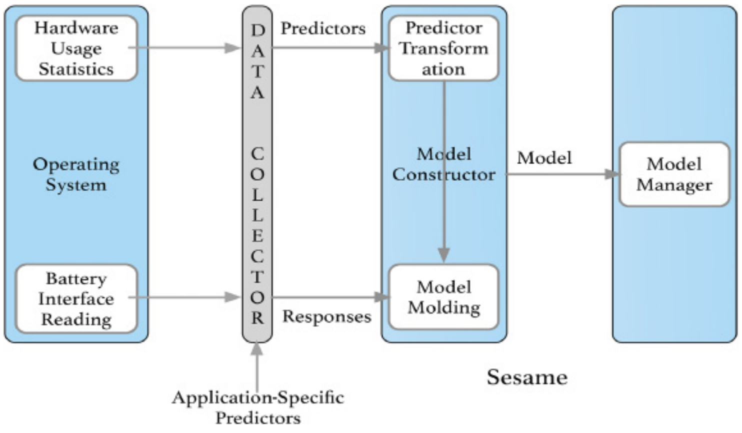

Fig. 8. Sesame architecture for automatic model generation.

update rates of the sources will generate error. For example, the processor's P-state (P0–P5) residency is updated at 250Hz, and reading this P-state at 100Hz produces 20% error, as the data collector may miss intermediate state changes. In addition, there can be delay between the predictor value update and the actual task performed. Accessing some data sources, such as reading the battery interface, also introduces overheads. Sesame reduces these overheads by adapting access rate to the source update rate. If the source's update rate is higher than Sesame's update rate, then it polls. Otherwise, Sesame waits for the changes in the dependent sources. As well, Sesame introduces a new system call that can read multiple data structures of subcomponents at a time from the OS.

The model constructor is the heart of Sesame. To generate a model with improved accuracy and update rate, Sesame applies model molding. Model molding works in two steps. The first step is called *stretching*, which involves constructing a model of the highest accuracy with a lower update rate than the target rate. This accurate model is constructed by averaging several readings, and the rationale is based on the observation that accuracy is higher when the battery interface readings are averaged over a longer period of time. For example, if the battery update rate is 0.5Hz, then a 0.01Hz rate model will be constructed by averaging consecutive 50 battery interface readings and the accuracy will be higher. In the second step, the low rate accurate model is compressed to construct a high rate model, which is achieved by applying the linear regression coefficients in calculating the energy consumption for the desired time interval.

Since the energy consumption models depend on the usage of different subcomponents or predictors, such as CPU frequency and power states, several such predictors can be large. Therefore, it is a challenge to find the actual predictors. For that, Sesame applies transformation on the predictors using principal component analysis (PCA) [Smith 2002]. The model molding and PCA together improve the accuracy of the model. However, the model constructor may generate models for three categories of system configurations: (i) information about the hardware and manufacturer; (ii) software settings, such DVS enabled or disabled; and (iii) user controlled settings, such as brightness and volume. Finally, the model manager adapts the model to the runtime system configuration. It compares the energy readings from the active model with the low rate version of the model obtained from the battery interface. If the error is

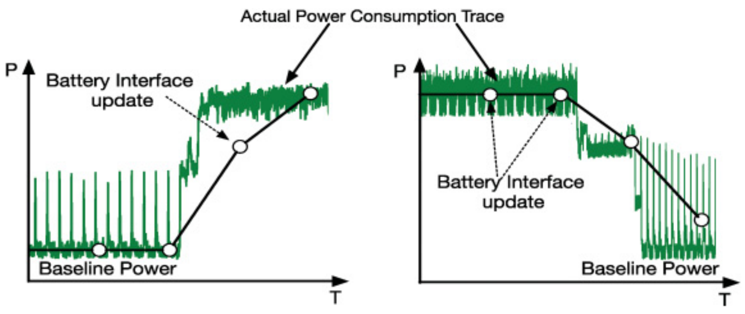

Fig. 9. Identification of the subcomponent power states by DevScope. The green line represents the power states of a subcomponent in a power trace, and the black line represents the power values for the power states realized by DevScope from the periodic battery interface update.

beyond a threshold, the model constructor generates a new model with new predictors. In this way, the model manager adapts to the system usage.

6.5. DevScope

DevScope [Jung et al. 2012] also uses battery interface updates to overcome the practical measurement requirements. Subsequently, it faces a similar challenge to Sesame—the low update rate of the battery interface. DevScope has four components: a battery update event monitor unit, a timing controller, a component controller, and a power model generator. DevScope first finds the hardware components available in the system and their configurations. The battery event monitor collects the discharging current information. The timing controller unit estimates the update rate of the battery interface. It also informs the component controller when to begin and terminate a test. The component controller generates and performs component specific test scenarios accordingly. Finally, the power model generator analyzes the test results and generates the power model coefficients.

Since the update rate of the battery interface can vary and cannot be controlled, DevScope adopts an event-driven approach that considers the battery update as an event. The battery monitoring unit keeps a timestamp record for every such event, and the timing controller finds the update rate from these timestamps. After calculating the update rate, the timing controller triggers a test scenario; in this way, the test scenarios are synced according to the battery interface update.

DevScope considers another challenge in recognizing the power states of the subcomponents, such as Wi-Fi and cellular network interfaces. However, recognizing state transition is difficult and requires the knowledge about the durations of different power states. In some cases, these transitions are governed by the workload and operating conditions. Consequently, DevScope employs different workloads repeatedly to determine the threshold that causes the power state transition and update in the battery interface. Figure 9 shows the actual power states of a subcomponent and the measurements realized by DevScope.

6.6. AppScope

The authors of DevScope proposed an application framework for energy profiling of the applications in mobile devices: AppScope [Yoon et al. 2012]. It depends on the DevScope power models. The power profiling of AppScope works in three phases. In the first phase, AppScope requests the detected hardware components. The second step involves the analysis of the usage of these subcomponents and their status changes.

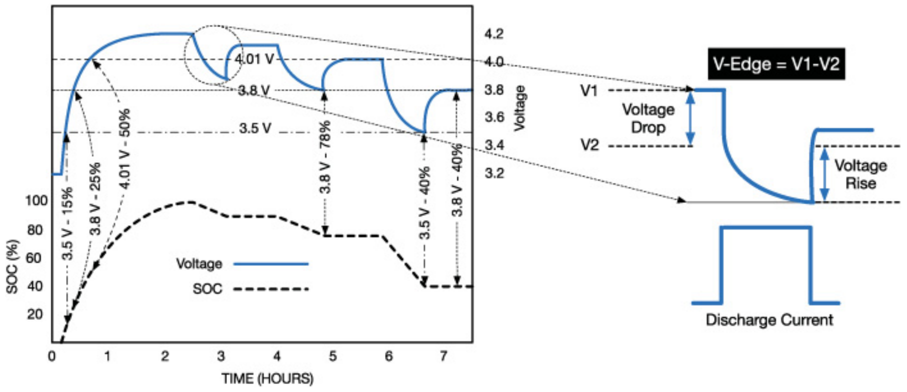

Fig. 10. The SOC and instantaneous voltage plots in the left figure show that the same voltage can represent multiple SOCs. The right figure illustrates how V-edge detects discharge current from the instantaneous change in voltage.

Finally, the energy consumption of an application is estimated by summing up the usage statistics of the subcomponents used by the applications.

One common limitation of the earlier described approaches is that they cannot isolate the usage of shared resources, such as display usage per application. AppScope does not rely on the system files for usage statistics of such shared subcomponents. Rather, it uses Android RPC with a binder RPC protocol [OpenBinder 2005; Schreiber 2011] and other debugging tools such as Kprobe [Keniston et al. 2011]. Kprobe monitors the events related to hardware component requests and analyzes the usage statistics that are required for applying power models. Subcomponent-specific functions are used for collecting data. For example, Linux CPU Governor interface is used for collecting the CPU usage and frequency information. The Wi-Fi usage is detected from the lower layer function calls in the Linux kernel. Then the power state is determined using the packet rate, and the energy consumption is computed from the active time duration of the interface. In the case of 3G, the RPC binder is used for detecting the hardware operation. The changes in the network connectivity are detected based on the radio interface layer interprocess communication data. In the case of display, the AppScope uses Android ActivityManager to find the application running in the foreground and tracks that activity until another activity is brought in the foreground or the display is turned off. Finally, the usage of GPS is tracked through the calls to the LocationManager. AppScope counts such calls when the GPS is activated and distributes the energy consumption according to the number of calls by different applications.

6.7. V-Edge

Similar to the previously described on-device power profilers, V-edge [Xu et al. 2013] aims to generate power models through self-metering. Its working principle is close to PowerBooter. The major difference with PowerBooter and other SOC-based approaches is that V-edge uses the changes in the instantaneous terminal voltage of the battery to infer current draw, whereas the SOC-based methods avoid such instantaneous voltage drop to reduce the SOC estimation error. Figure 10 shows how the instantaneous voltage can mislead the SOC estimation and how V-edge exploits it. However, the main effect of this difference is a speedup in power model generation compared to PowerBooter. To understand why, recall that PowerBooter needs to keep the phone in a particular constant power state long enough so that the SOC value changes, whereas V-edge can measure the current almost instantaneously. Therefore, any OCV-based battery model, such as the Rint or Thevenin model, can be plugged in to infer the discharge current (see Section 3.2). V-edge effectively utilizes the current draw across the internal, R, resistor. The equations of the Thevenin model can be written as

$$\begin{aligned} V_t &= OCV - V_c - \triangle I \times R \\ V_t &= OCV - V_c - V_{edge} \\ V_{edge} &= R \times \triangle I = R \times I - R \times I_0 \\ I &= \frac{1}{R} \times V_{edge} + I_0. \end{aligned}$$

When there is noticeable amount of current change, OCV and V_c remain the same for a short period of time. In Figure 10, we notice that there is a sharp change in the voltage when there is a current draw. This instantaneous voltage drop is caused by the internal resistance. After that, the voltage drops slowly because of the current discharge on the battery. The instantaneous voltage drop ($R \times \triangle I$) is defined as V-edge in the preceding equation. I_0 refers to the baseline current consumption that can be achieved by starting all of the training from the same baseline while generating the power models.

To find such V-edges, Xu et al. [2013] applied an approach similar to DevScope. First, a mobile device is kept idle for a longer period than the battery interface update interval. Then the CPU utilization is increased for a while and the voltage is sampled at 1Hz. The CPU is kept idle when there is a voltage drop. The sampling is stopped when there is an update in the voltage. The interval between these two voltage drop and update incidents facilitates the detection of V-edge and thus the current measurement as illustrated in the figure.

The system architecture of V-edge consists of a data collector, model generator, and power profiler. The data collector collects battery voltage information for generating the power model and the utilization statistics of different subcomponents for estimating the power consumption of applications, such as CPU, screen, Wi-Fi, and GPS. The component-specific power models are built on top of the V-edge by running a set of corresponding training programs. The training begins from the initial power state of the subcomponents to ensure the consistency in their V-edge values. Finally, the power estimation is done using the collected resource utilization statistics.

6.8. Summary

These on-device profilers rely on the smart battery interface updates for power consumption measurements. They do not require any external device or calibration, except the NEP. The NEP uses certain feature calibration, but it still belongs to this category, as it does not require any external device measurements. Most on-device profilers use simple linear regression models, except the NEP and Trepn, and their model generation is conducted automatically. They profile the energy consumption of subcomponents, and the whole device as well, while running applications. For that, they rely on the support of the OS for collecting the utilization statistics of each component.

Every profiler is unique in some aspects. For example, the NEP can trace the cellular network interface connectivity status in addition to power consumption, which makes the NEP different from other tools. Trepn requires the support of special component-specific special sense resistors and power management IC. PowerBooter is the first of the on-device model-based profilers, which depends on the changes in the SOC and thus takes a very long time for model generation. Sesame, on the other hand, directly reads from the battery interface, and thus its model generation takes shorter time than PowerBooter. DevScope and AppScope also rely on the current readings. However, DevScope is unique in the way that it recognizes the power states from the

Table III. On-Device Energy Profilers without Offline Support and Their Comparison

Name/ Author	Profiling Coverage	Model Type	Battery Interface Reading	Accuracy
NEP	On-device stand-alone profiler	Not Applicable	Voltage, current	99%
Trepn	Profiler for device and subcomponents	Not applicable	Current	Close to NEP or Monsoon
PowerBooter	Profiler for device and subcomponents	FSM, linear regression	SOD	96%
Sesame	Profiler for device and subcomponents	Linear regression, utilization	Current	86% (1Hz) & 82% (100Hz)
DevScope	Profiler for device and subcomponents	FSM, linear regression model, and utilization	Current	95%
AppScope	Profiler for component usage and energy consumption of applications	Built on DevScope, linear model	Current	92%
V-edge	Model for device and subcomponents	Utilization, linear model	Instantaneous voltage	86%

battery updates. AppScope depends on the DevScope power models and emphasizes the energy consumption of the applications using shared resources. Unlike the other profilers, AppScope uses an Android RCP binder and Kprobe for collecting fine-grained component use statistics by the applications. Finally, V-edge uses instantaneous voltage drop to measure the current draw from the battery and generates a model faster than PowerBooter.

With respect to accuracy, the NEP and Trepn provide the highest measurement accuracy. The NEP measures energy consumption within the range of 3mA, and thus its accuracy is close to the measurement with the Monsoon Power Monitor–like devices [Microsoft 2010]. In the case of Trepn, we assume the highest accuracy, as it senses current draw directly from the component-specific sense resistors. Since only voltage can be sensed across a sense resistor, Trepn suffers from offset current and ADC conversion error. The rest of the profilers depend on battery interface updates, and the update rate poses a potential challenge in minimizing the error rate and improving the accuracy of the power models. Sesame acknowledges this issue by taking an average over a number of samples. In Table III, we can see that the accuracy of Sesame is higher with model constructed at lower rate and that the profiler suffers from 14% error with the models with a higher update rate. This error is mostly caused by the extra access overhead. On the contrary, DevScope proposes to synchronize the smart battery update events with the subcomponent tests to deal with the slow update rate of a smart battery interface.

7. ON-DEVICE PROFILERS WITH OFF-DEVICE MODEL CONSTRUCTION

Unlike the pure on-device profilers described in Section 6, these applications require offline calibration and power measurement phases. Sometimes these applications are developed by the mobile vendors and come as an integral part of the mobile systems. In this section, we describe the power profilers that belong to this category, and at the end we summarize their key similarities and differences.

7.1. Android Power Profiler

The Android OS has its own energy models and profilers for estimating the energy consumption of different applications and components. The complete profiling system is based on three subcomponents: BatteryStats, a list of power consumption values of the hardware components (power profile), and the power models. The BatteryStats depends on the power profiles and models to estimate the power consumption. We examine this service in detail in the following sections.

1. BatteryStats. BatteryStats is responsible for tracking the usage of hardware components, such as CPU, GSP, Wi-Fi, wakelocks, GSM radio scanning, phone (call), and Bluetooth. The usage information of these components are recorded along with the timestamp. BatteryStats does not directly measure the energy draw from the battery. Rather, it calculates the total utilization of different hardwares from the timestamps and estimates the energy consumption. BatteryStats collects the statistics in two different ways: different services push the component state changes to the BatteryStats, or it collects the information of CPU and other components used by the applications periodically from the `proc` system files. BatteryStats stores the statistics for 30 minutes so that data is not lost when there is a sudden reboot or failure of the system. Most of the other power profilers, such as PowerBooter and AppScope, receive these statistics from the BatteryStats or directly retrieve them from the `proc` system files. BatteryStats also serves these statistics to other requesting services. Therefore, registering with BatteryStats is safer, as the locations of the stat files can be different in different devices. The recent Android Lollipop provides a tool to extract the BatteryStats from mobile devices for off-device analysis with Battery Historian [Android 2014b].

2. Power profile values. To estimate the energy or power consumption, BatteryStats depends on premeasured power values of different hardware components. These essentially are pretrained power model coefficients. The values come with the Android framework application and are stored in an xml file [Android 2014a]. The file contains information about the power states supported by the CPU, GSM radio, Wi-Fi, display, Bluetooth, and audio and video DSP unit. The file also includes the current drawn, in milliamperes, by these components at those states, which are very specific to the corresponding device models provided by the manufacturers. For example, the file contains the clock speeds supported by the CPU and the current drawn at each clock speed. The Android Power Profiler assumes that all of the cores share homogeneous frequency and power consumption characteristics [Android 2014a].

3. Power models. Once the subcomponent usage statistics and the basic power drawn by them are known, BatteryStats can easily compute the energy consumed using some basic models. In Table IV, we present a list of power models used by BatteryStats in Android devices, which we extracted from the Android framework source code. We can see that these are simple utilization-based models. BatteryStats first computes the time span of the hardware resource utilization and then computes the energy consumption according to the power states. For wireless communication, it first calculates the transmission or reception speed in bytes per second and then calculates the energy per byte. Finally, the system attributes the energy consumption to an application simply by summing up the energy drawn by the components utilized by the application.

Some of the resources can concurrently be used by multiple applications, and BatteryStats makes an additional effort to distribute the cost between those applications. In this case, wakelocks are useful, and a wakelock for a component can be set by more than one application. After that, those applications that have set the wakelock share the power cost. The amount of time the applications are tied up with the wakelock determines their partial costs.

7.2. PowerTutor

PowerTutor [2009] is a power profiling application, developed based on the PowerBooter model, for Android mobile devices. However, the model depends on off-device power

Table IV. Android Subcomponent- and Application-Specific Energy Profiling Models

Subcomponent/ Application	Statistical Variable	Models
Screen	Time spent at brightness level i, T_{bri-i}	$E_{Screen} = \sum_{i=1}^{N}(P_{brightness} \times T_{bri-i})$
System Idle	The total duration T_{total}; time spent when screen is on, $T_{screenOn}$	$E_{Idle} = P_{cpuIdle} \times (T_{total} - T_{screenOn})$
Radio (Cell Standby)	Time spent when signal strength is i, T_{str-i}; total time spent in scanning, T_{scan}	$E_{mobileStandby} = (\sum_{i=1}^{N} P_{strength} \times T_{str-i}) + P_{radioScan} \times T_{scan})$
Phone (Call)	Duration of a call, i, T_{call-i}	$E_{call} = \sum_{i=1}^{N}(P_{call} \times T_{call})$
Bluetooth	$T_{bluetoothOn}$, $Ping_{count}$	$E_{Bluetooth} = (P_{bluetoothOn} \times T_{bluetoothOn}) + (Ping_{count} \times P_{atCommand})$
Wi-Fi$_{App}$	Total duration an app, i, uses Wi-Fi, $T_{wifiApp-i}$; scan time for the app, $T_{wifiScan-i}$	$E_{wifiApp} = T_{wifiApp-i} \times P_{wifiOn} + T_{wifiScan-i} \times P_{wifiScan}$
Wi-Fi$_{noApps}$	Total Wi-Fi usage time $T_{wifiGlobal}$; Wi-Fi usage time by an app, i, $T_{wifiApp-i}$	$E_{wifinoApps} = (T_{wifiGlobal} - \sum_{i=1}^{N} T_{wifiApp-i}) \times P_{wifiOn}$
CPU$_{App}$	Time spent at speed, i, $T_{speed-i}$; time spent in executing app code, $T_{appCode}$; time spent to execute system code, $T_{sysCode}$	$E_{cpuApp} = \sum_{i=1}^{N} \frac{T_{speed-i}}{\sum_{i=1}^{N} T_{speed-i}} \times (T_{appCode} + T_{sysCode}) \times P_{speed-i}$
Wakelock	Wakelock ime, $T_{wakeLock}$	$E_{wakeLock} = (P_{wakeLock} \times T_{wakeLock})$
GPS	GSP usage time, T_{gps}	$E_{gps} = (T_{gps} \times P_{gps})$
Mobile Data (Byte/Sec)	Radio active time, $T_{radioActive}$	$mobileBps = (mobileData \times 1000/T_{radioActive})$
Wi-Fi Data (Byte/Sec)	Wi-Fi active time, $T_{wifiActive}$	$wifiBps = (wifiData \times 1000/T_{wifiActive})$
Average Energy Cost per Byte		$E_{byte} = (\frac{P_{wifiActive}}{wifiBps} \times wifiData + \frac{P_{radioActive}}{mobileBps} \times mobileData)/(wifiData + mobileData)$
App		$E_{App} = E_{cpuApp} + E_{wakeLock} + E_{wifiApp} + E_{gps} + (tcpBytesReceived + tcpBytesSent) \times E_{byte}$

measurement of subcomponents. The application estimates the power consumption of different hardware components and applications. PowerTutor illustrates the share of energy consumed by the display, CPU, Wi-Fi, GPS, and 3G using a pie chart. It also uses line graphs to describe the runtime power consumption of these components in joules. For that, PowerTutor depends on fine-tuned power measurements of the components at different power states and on the usage statistics collected from the `proc` system files and the Android BatteryStats.

Similar to the Android Power Profiler, PowerTutor also estimates application-specific energy consumption. This enables the application developers to visualize the energy consumption of their applications and thus enables further optimization. At the same time, the users can understand the impact of their interaction on the battery life of their mobile devices. However, it is challenging to estimate the power consumption of those applications when more than one application is running at the same time and using or sharing common resources. In these situations, it is not clear how to divide the hardware energy consumption. PowerTutor solves this by estimating the cost for a single application, requiring that it has been the only application running at that time.

7.3. PowerProf

PowerProf [Kjaergaard and Blunk 2012] is an unsupervised power model generation mechanism that applies a genetic algorithm. It requires a set of training measurements to build the power models. The training can be initiated when the device is idle or even during the installation process of the application.

PowerProf system architecture consists of measurement data collection through the NEP APIs. Then, the genetic algorithm, hosted in a separate computer, crunches this data to generate the power models. The steps involved in the process are as follows. At first, a request is sent to the battery interface for providing power measurements with timestamps. The specific phone features are exercised, and the corresponding timestamps are logged for the beginning and ending of the exercise. Then, some obvious characteristics are determined, such as the background power consumption. Next, the genetic algorithm is applied to find the optional parameters required for the power models. The fitness function of the algorithm calculates the distance between the power consumption measured by the API and the model. Finally, the parameter values that minimize the fitness function are used in the final model. The final power model is a conditional function of four time parameters, which resembles four power states of an individual component.

7.4. Summary

We have found that the Android system energy profiler and PowerTutor depend on premeasured power consumption values. In the case of the Android Power Profiler, the component-specific power values come from the vendors. However, it is possible to have incorrect power measurement values in the power profile file, which may provide misleading estimates of the energy consumed by the applications or devices. For instance, we notice that the battery sizes of two devices may be claimed to be the same in the power profile files although they are different from each other.

The other profiler, PowerTutor, has more component and application coverage than the Android Power Profiler. For power consumption of individual hardware components and their state timer values like Wi-Fi, 3G, and others, PowerTutor depends on premeasured constant values. Therefore, the accuracy of PowerTutor is 97.5%. At the beginning, the target devices for power models were HTC G1, G2 and Nexus One devices. However, it also works on other devices with rough estimates, as the power consumption of the hardware components varies. PowerProf stands out from all of the on-device profilers in applying genetic algorithms to build the power models.

8. OFF-DEVICE ENERGY PROFILING IN A LABORATORY

In this section, we examine off-device energy profilers. They profile resources utilized by applications, perform code analysis of the applications in a device or in an emulator, and then map such activities to energy consumption with external power measurement tools to generate power models. This method typically supports fine-grained and accurate characterization of the energy consumption of the target application, subsystem components, or the device. Consequently, they are suitable for debugging applications.

8.1. PowerScope

PowerScope [Flinn and Satyanarayanan 1999] is one of the early energy profilers. It uses both off-device and on-device profiling of the application. The profiling of the applications and data collection take place on-device. The energy profiling is done off-device. Two on-device components, a system monitor and an energy monitor, share the responsibility of data collection and run in two different systems. The system monitor is hosted in the profiling computer system. The sample collection is triggered by a digital multimeter. The profiling sample consists of the program counter and the IDs of the running processes. The profiler also records some additional information, such as whether the system is handling any interrupt or not, the path name associated with the execution of the current process, and loading of the shared libraries.

The energy monitor runs in the data collection system. It communicates and configures the digital multimeter to sample the current drawn from the external power source. The output of the on-device profiling stage is a sequence of current samples and a correlated sequence of program counter and process identifier samples. Next, the off-device component, the energy analyzer, generates activity-based energy profiles by integrating the product of the supply voltage and instantaneous current over time. The current values I_t are sampled at a periodic interval of Δt. Since the supply voltage from an external source is almost constant, the energy over n samples using a single voltage measurement value, V_m, is given by

$$E \approx V_m \sum_{t=0}^{n} I_t \Delta t. \qquad (4)$$

This technique requires the drawn current to be sampled with intervals of Δt. However, the energy analyzer then reads the data collected by the monitors and correlates each system sample with the current sample. If the system sample belongs to a process, it assigns the sample to a process bucket using the process ID value. For an asynchronous interrupt, the sample is assigned to an interrupt specific bucket. If no process has been executing, the taken sample is attributed to the kernel bucket. Then energy usage for each process or interrupt is calculated using the previous equation. Finally, it generates an energy profile consisting of the CPU time, the total energy, and the average energy consumption of each process and corresponding procedures.

8.2. Joule Watcher

Joule Watcher [Bellosa 2000] modified the context switch routines and data structures in the Linux kernel to record the values of HPCs. The relation between the number of events and power consumption is linear. The profiling system first generates microbenchmarks of power consumption for four kinds of events: microinstruction execution, floating point operation, layer 2 cache access, and main memory access. The energy consumption is determined with an external power monitor device. Then a regression-based power model is built based on the microbenchmarks. This approach is limited for three reasons. First, the number of events that can be profiled is limited by the number of counters, which in turn depends on the architecture. Second, the power model needs to be trained for each device type and configuration in the lab. Third, the energy per event is not constant and depends on the clock speed. Therefore, the benchmarking requires careful reconfiguration of various speed levels for different types of CPU events.

8.3. Fine-Grained Profiling with Eprof

Eprof [Pathak et al. 2012] is an off-device energy profiling framework for Android- and Windows-based mobile devices. The framework consists of a few components: the routine or system call tracer, the energy profiler, and the profile viewer. Android supports application development with the software development kit (SDK) and native development kit (NDK) for developing the critical parts of the application. For SDK, Eprof instruments the default routine profiling framework to consider only caller-callee invocations. It also performs periodic sampling and the corresponding sampling timestamp. This reduces the tracing overhead. NDK calls are traced at the C library interface. To trace system calls, Eprof instruments the Android framework to log the time, system call parameters, and call stacks.

The logged traces are postprocessed to account for the energy consumption. The system calls are the indication of different hardware utilization by the applications, and such calls are mapped into FSM models developed in Pathak et al. [2011]. Power consumption of a hardware component in a state is constant, and a component can have only one power state at a time. Tracing system calls serves two purposes in this framework. First, they can clearly trace which components are requested by the application and how long that resource is being utilized. Second, the system call can be retraced back to the callee and thus the energy consumption of a routine or a function.

8.4. Banerjee et al. [2014]

Similar to Pathak et al. [2012], Banerjee et al. [2014] also worked on a profiler toward identifying the energy anomalies. In this section, we briefly describe the energy profiling mechanism. The energy anomaly findings are covered in Section 9.2. Banerjee et al. [2014] developed a test framework that profiles the energy consumption in three steps. First, the flow of events in an application is traced and an event flow graph (EFG) is generated using Hierarchy Viewer [Android 2014c] and Dynodroid [Machiry et al. 2013]. Hierarchy Viewer provides information about the execution of the UI elements in an application, and a sequence of events is generated when Dynodroid interacts with the application. However, Dynodroid does not generate the flow graph of the events by itself, and consequently it was instrumented. When the EFG is ready, a set of event traces is generated in the second step. The length of such traces can be arbitrary and must start from the root UI. Akin to Eprof [Pathak et al. 2012], this framework also relies on the system calls to identify the hardware component usage, and likewise they are recorded while executing the event traces by instrumenting the applications in the third step.

Unlike Eprof, the framework proposed by Banerjee et al. [2014] depends on utilization-based power models that are developed off-device based on the same power measurements used by the Android Power Profiler (see Section 7.1). All of the event graph generation, tracing, and instrumentation of mobile applications are done on an emulator in a desktop computer.

8.5. Shye et al. [2009]

Shye et al. [2009] developed an energy profiler for smartphones. The profiler estimates the energy consumption of the system and different components of mobile devices. The main idea here is that a user-space Android application collects system usage information and other performance statistics, then uploads this information to a remote server. The data is analyzed for identifying a usage pattern for building power estimation models for mobile architecture with an abstraction of two distinct power states: active and idle. It was assumed that the screen would be ON or the system wakelock would be occupied by the application while the screen would be OFF. Although energy consumed by mobile devices varies with the workload, the energy consumption is relatively invariant in the idle state. Shye et al. [2009] applied different workloads and then used the generated logs to produce the power models. The main limitation of the system is that it requires device-specific calibration.

8.6. Summary

Table V compares the off-device energy profilers discussed in this section. Among them, only Joule Watcher depends on HPCs to profile thread-level energy consumption. On the other hand, PowerScope, Eprof, and Banerjee et al. [2014] apply code analysis to estimate the energy consumption of functions and applications. PowerScope relates the execution time of a process with the current drawn during the executing period, whereas Eprof and Banerjee et al. [2014] trace system calls in the functions. Then, they apply FSM- and utilization-based models respectively to estimate the energy consumption of the hardware components corresponding to the system call and relate

Table V. Off-Device Energy Profilers and Their Performance Comparison

Name/Author	Profiling Granularity	Measures	Model Type	Accuracy
PowerScope	Device and process levels	Current, voltage	Code analysis and current integration over time	Not reported
Joule Watcher	Thread level	Thread energy consumption	Utilization, linear regression	Not reported
Eprof	Device, subcomponent, and application levels	System calls, energy	Code analysis, FSM	94%
Banerjee et al. [2014]	Device, subcomponent, and application levels	System calls, energy	Utilization	Not reported
Shye et al. [2009]	Device and subcomponent levels	Energy	Current integration over time, linear regression model	93%

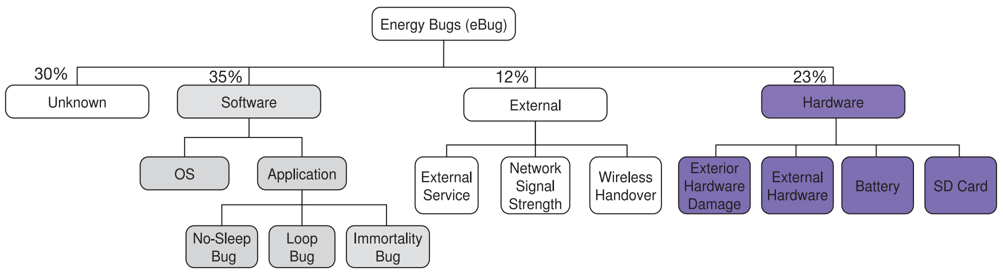

Fig. 11. An overview of energy bugs.

the energy consumption with the callee function. The energy profiler implemented by Shye et al. [2009] also depends on a simple utilization-based model.

With respect to accuracy, the performance of PowerScope depends significantly on data sampling frequency and the external device monitor. The performance of Joule Watcher is limited by the CPU architecture and power consumption, as the hardware events may not be constant, as mentioned earlier. Apparently, the FSM-based Eprof seems to be more accurate with an accuracy of 94%. Although Banerjee et al. [2014] did not discuss the accuracy of their profiler, it is likely to suffer from more error than FSM-based Eprof. The profiler from Shye et al. [2009] is also less accurate than Eprof.

9. ENERGY DIAGNOSIS ENGINES

In addition to developing energy profilers, researchers have sought to better understand the energy consumption behavior of different applications, particularly to detect program code that causes suspiciously high energy consumption. In this section, we first discuss a taxonomy of energy bugs for mobile devices and then present several energy debugging tools.

Energy bugs. Pathak et al. [2011] mined online blogs and identified several energy bugs from reports from users. Figure 11 shows a hierarchical taxonomy of the energy bugs for mobile devices. Although the list may not be exhaustive [Pathak et al. 2011], it allows us to note that energy bugs can be divided into four categories. The software-related bugs are divided further into OS and application types. The applications suffer from three types of bugs: no-sleep, loop, and immortality bugs. The no-sleep bugs are the results of software coding errors that fail to release a wakelock and prevent

Table VI. Classification of the Energy Diagnosis Engines

Name/Author	Profiling Granularity	Model Type	Model Construction	Deployment Type
Eprof	Device, subcomponent, and application levels	Code analysis and FSM based	Off-device	Off-device
Banerjee et al. [2014]	Device, subcomponent, and application levels	Linear regression	Off-device	Off-device
eDoctor	Device and subcomponent levels	Not applicable	Not applicable	On-device
Carat	Application level	Statistical	Off-device	On-device

the device or one of its components from switching to a sleep mode. If a component-specific wakelock, such as WiFiLock [Android 2015], is not released by the application before exiting, the component continues to be in a high power state [Banerjee et al. 2014]. With a loop bug, an application executes some unnecessary code again and again. In the case of immortality, a buggy application is killed by the user, but the application restarts again with the buggy nature. External energy bugs are triggered by the wireless network conditions. A device increases the transmission and reception power of the wireless radio, when there are poor signal strengths. Poor network status may also trigger frequent wireless handovers, which also depletes the battery faster. The hardware bugs are related to the faulty electronics, such as the battery, charger, exterior hardware damage, and external peripherals like the SD card. A faulty battery may drain very fast. Such a scenario can arise when the phone is charged with a faulty wall or USB charger. External damage to the device may cause the home screen or power button to be very sensitive, which may result in frequent display of ON/OFF. In addition, writing to a corrupted SD card may drain the battery quickly if the device tries to write in a loop.

A list of energy debugging tools are presented in Table VI, and we can see that these tools, such as Eprof, can be used to understand the energy consumption behavior of the applications. In general, these systems aim to answer one or more of the following questions. How much energy consumption should be normal for an application or a component? Does the abnormal energy consumption stem from poor system configuration or user behavior? Would changing the system setting improve the energy savings and by how much?

9.1. Diagnosing with Eprof

Considering the debugging challenges for different kinds of energy bugs for mobile devices, Pathak et al. [2012] proposed the Eprof energy debugging framework, which aims for fine-grained energy consumption analysis of applications and the OS in mobile devices. Eprof finds the energy hotspots in the source code by instrumenting both the OS and applications, then traces system calls. In this way, Eprof [Pathak et al. 2012] does fine-grained profiling by identifying the contribution of code in total energy consumption. For example, *lchess* spends 30% of total energy in checking user moves in the game and 27% of the energy in code offloading [Cuervo et al. 2010]. Eprof also finds energy bugs in the application. It specifically looks for the acquisition of wakelocks and their releases in the code.

9.2. Banerjee et al. [2014]

Banerjee et al. [2014] also classified mobile applications according to their energy usage: energy bugs and hotspots. The application energy bugs are similar to those identified by Pathak et al. [2011] as shown in Figure 11. On the other hand, the hotspots include the applications that execute networking code in infinite loops, the applications that heavily depend on very high sampling rates at the background, and the applications that suffer from tail energy or suboptimal resource binding. The suboptimal resource binding refers to binding or releasing the resources too early, which causes the hardware to be in high power states longer than actually required.

To identify energy bugs and hotspots, Banerjee et al. [2014] relied on their offline profiling framework (discussed in Section 8.4). The authors divided an event trace into four phases (PRE, EXEC, REC, and POST) and generated a corresponding energy utilization ratio[1] trace. In the PRE stage, the device remains in the idle or very low power consuming stage. EXEC refers to an event execution stage. In the REC phase, the device consumes tail energy before going to the completely idle state in the POST phase. To detect the energy bugs, the energy utilization in the PRE and POST phases are compared. If the difference is more than a threshold value of 50%, then those are marked as buggy applications. On the other hand, energy hotspots are screened during the EXEC and REC phases in an event trace using a technique that employs discords [Keogh et al. 2005] to find the abnormality in the energy utilization in a subsequence by comparing it to the remaining subsequences in the EXEC and REC stages.

9.3. eDoctor

eDoctor [Ma et al. 2013] is another energy debugging tool. This application runs in the user space just like other applications and investigates multiple phases during the execution of other programs. eDoctor then identifies the phases that have strong correlation with the energy waste or abnormal energy usage. The execution phases of an application are then mapped to the execution intervals. In each interval, an application consumes a certain type of resources. Therefore, an anomaly can be detected when an application deviates from the normal behavior. The anomaly detection can be fine grained by correlating such behavior to the system configuration.

eDoctor consists of four components: data collector, analyzer, diagnosis, and prognosis advisor. The data collector finds the resources, such as CPU, GPS, and sensors, used by the applications. At the same time, it uses energy models to estimate the energy consumed by these hardware components. As the energy consumption of these components also depends on their power states, eDoctor records their state changes as events just as the Android BatteryStats does (Section 7.1). The analyzer analyzes the resource usage over time. From the data, eDoctor generates phase information and energy consumption for each application. eDoctor also constructs a phase table for each application using a k-means clustering algorithm. The diagnosis engine identifies the energy-buggy applications in two steps. First, it finds the applications with energy-heavy phases and then finds the corresponding events. However, eDoctor analyzes a complete trace between two consecutive charge intervals.

Finally, the prognosis advisor recommends three actions to a user. First, if the present version of an application consumes more energy, eDoctor recommends the user to switch to the earlier version. Second, if an application continues running even after the user stops using it, eDoctor suggests that the user kills the application manually. Third, eDoctor recommends the user to change the settings, such as reducing the brightness level of the display or turning off the GPS, in the case of overusing a hardware component.

[1]The energy/utilization ratio expresses the energy efficiency of an application over a period of time. A high ratio implies an inefficient application.

9.4. Carat

Akin to eDoctor, Carat [Oliner et al. 2013] is an energy anomaly detection application. Carat assumes that there is no prior information on how much energy an application should consume and whether such energy consumption is abnormal. Consequently, Carat depends on the community of devices and applies a collaborative approach. It collects the name and number of applications running in mobile devices and their battery interface information, provided by the device battery manager, according to the update rate. From the battery interface reading, Carat computes the discharge rate for different configurations and running applications. Since Carat depends on the community of devices, it can compare the energy usage of an application among the devices. In this way, Carat classifies the applications into two categories: hogs and bugs.

The comparison is straightforward for energy hogs, for which two energy rate distributions, with and without the application across the device community, are compared. Carat uses a 95% confidence standard error of the mean (SEM) as error bars around the means of the distributions to detect abnormal energy behavior. If the distance of the means including the error bars is greater than zero, an application is marked as a hog. The detection of applications with energy bugs requires that a similar comparison is made for each application on a device. More specifically, an application is an energy bug if the energy rate distribution distance including the error bars with the application on the current device being analyzed and on other devices is greater than zero.

Carat performs further diagnosis of an application with different system settings, such as Wi-Fi active or inactive, user roaming or stationary, and the OS version. Like the eDoctor system, Carat recommends actions to the user to increase the runtime battery life. Those recommendations include killing or restarting an application, and even changing the OS version. In addition, Carat also informs user about the battery life improvement for the corresponding action. However, Carat does not help during the application development process.

9.5. Summary

The diagnosis tools, namely Eprof, eDoctor, Carat, and the method proposed by Banerjee et al. [2014], aim to identify energy hotspots and bugs inside the applications and characterize the applications as energy hoggy or energy buggy. The definitions are quite similar in all cases. Eprof and Banerjee et al. [2014] trace system calls and depend on their own profilers to find the energy hotspots and bugs in applications. Banerjee et al. [2014] and eDoctor analyze the application execution in multiple stages. However, Carat differs from the others in the aspect that it studies energy consumption behavior of applications across a community of devices. The tool from Banerjee et al. [2014] considers suboptimal resource binding while finding hotspots, which is not considered in Eprof and others.

10. ENERGY PROFILING ACCURACY AND RECOMMENDATIONS

Table VII shows that the accuracy of different profilers varies between 86% and 97.5%. Although this measure is often considered as the most important attribute of power profilers to evaluate, it is difficult to promote a particular profiler above the others based on the reported accuracy. The reason is that the evaluation of their accuracy can be biased toward those applications that use only the modeled subcomponents. In addition, some profilers may trade some accuracy for other desirable features. For example, PowerBooter takes longer time for model generation but it is reported to provide high accuracy. On the other hand, V-edge and Sesame allow generating power models much faster but may be slightly less accurate.

Table VII. Component-Level Support of the Profilers and their Reported Accuracy. The Accuracy is Estimated against the Power Measurement Results with the External Devices

Profiler	Disp	CPU	GPU	GPS	BT	Wi-Fi	3G	4G	Cam	SD Card	Audio	Reported Accuracy
Trepn	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	99%
V-edge	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	86%
BatteryStats	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗	Not reported
PowerBooter	✓	✓	✗	✓	✗	✓	✓	✗	✗	✗	✓	96%
PowerTutor	✓	✓	✗	✓	✗	✓	✓	✗	✗	✗	✓	97.5%
DevScope	✓	✓	✗	✓	✗	✓	✓	✗	✗	✗	✗	95%
AppScope	✓	✓	✗	✓	✗	✓	✓	✗	✗	✗	✗	92%
Sesame	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	86%
Eprof	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓	✗	94%
Banerjee et al. [2014]	✓	✓	✗	✓	✗	✓	✓	✗	✗	✗	✓	Not reported
Shye et al. [2009]	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	✗	93%

The profilers differ in the number of subcomponents and their power states modeled, modeling methodology, rate of data collection, and power measurement techniques. All of these obviously contribute in one way or another to the overall accuracy of the profiler. For example, AppScope reports an error of 7% because DevScope does not model the power consumption of GPU. Still, their modeling approach, which is based on component-specific FSM, seems to perform more accurately than V-edge, which relies on simple linear models.

Although the contribution of some of the factors may be obvious, distributing the amount of total error among all the preceding factors is challenging. Such work requires benchmarking the profilers, which is difficult because some of the profilers exist only as research prototypes and their software is not openly available. Therefore, we choose to look into the evaluation techniques of the profilers to try to understand the extent of the preceding factors in contributing to the accuracy. At the same time, we also discuss the methods to limit the error due to the contributing factors.

10.1. Subcomponents and Their Power States

We looked into the accuracy evaluation methods of the profilers. In all cases, the power estimates provided by the profilers have been compared to measurements provided by external instruments. The reported accuracy of each of the surveyed profilers is presented in Table VII. In addition, they have usually been evaluated with applications that stress those hardware components that the profiler models. For example, the accuracy of PowerBooter was evaluated using applications (browser and YouTube) that require the subcomponents presented in the table. Consequently, if a user plays AngryBirds, which requires a GPU, PowerBooter will provide estimates with less accuracy, and such evaluation scenario was not reported. AppScope does not consider the power consumption of the GPU either and thus is reported to suffer from a 7% measurement error. The performance of V-edge was also evaluated with Gallery, AngryBirds, Skype, and browser applications. In the case of AngryBirds, it is likely that V-edge also suffers from similar error as AppScope, as it does not model the power consumption of the GPU. V-edge would also suffer from a larger error if it were tested with applications using the a cellular network, because it does not model the power consumption of those network interfaces. Therefore, if an application requires a nonmodeled subcomponent, the accuracy will degrade from the reported values and the amount of error depends on the power consumption characteristics of the nonmodeled subcomponent, usage, and power measurement techniques (see Section 10.4).

Table VIII. Example List of Component-Specific Models and Power Profilers

Author	Subcomponent	Model Type
Ma et al. [2009]	GPU	Linear regression
Hong and Kim [2010]		Utilization
Nagasaka et al. [2010]		Linear regression
Leng et al. [2013]		Utilization
Burtscher et al. [2014]		FSM
Tsoi and Luk [2011]	Heterogeneous multicore CPU	Utilization
Tudor and Teo [2013]	Homogeneous multicore CPU	Utilization
Zhang et al. [2013]		FSM
Rethinagiri et al. [2014]		Utilization (HPC)
Diop et al. [2014]	Multicore CPU and GPU	Utilization (HPC)
Garcia-Saavedra et al. [2012]	Wi-Fi	Fine-grained FSM
Ding et al. [2013]	Wi-Fi & 3G	Signal strength–associated FSM
Qian et al. [2011]	3G	FSM (Profiler ARO)
Hoque et al. [2014b]	Wi-Fi, 3G & 4G	FSM
Hoque et al. [2013b]		FSM
Lauridsen et al. [2013]	4G	FSM
Jensen et al. [2012]		FSM
Huang et al. [2012]		FSM
Dong et al. [2009]	OLED Display	Code analysis (associates a GUI object into pixels)
Chen et al. [2012]		Utilization
Dong and Zhong [2012]		Utilization and code analysis
Chen et al. [2013]		Utilization

Table VII shows that not necessarily all of the profilers cover every component available in mobile devices. For instance, only Eprof and Trepn consider the power consumption of the camera. In a recent study, Rajaraman et al. [2014] found that a camera consumes the same or more energy than the 3G or 4G interfaces, even in the focus mode. Although the power consumption of cellular network interfaces have been extensively studied [Siekkinen et al. 2013; Hoque et al. 2013b], the Android Power Profiler does not consider the energy consumption of cellular network interfaces. This applies to most of the on-device profilers as well. Table VIII lists some examples of component-specific power profiling and modeling work. Today, it is common for mobile devices to be equipped with multicore homogeneous cores. The energy efficiency of mobile CPUs recently has being enforced through new architecture with heterogeneous cores. Although there have been energy measurement and modeling for heterogeneous CPU cores [Diop et al. 2014] and GPUs [Leng et al. 2013; Burtscher et al. 2014], existing energy profilers focus mainly on single- or multicore CPU systems with homogeneous cores. The power consumption of homogeneous cores are equal while operating at a particular frequency, whereas the heterogeneous cores may have different power consumption characteristics.

Table VIII further shows that a significant number of component-specific models follow FSM. Among the power profilers discussed in this work, Eprof, AppScope, DevScope, NEP, and PowerProf also consider the tail state behavior of some components. Burtscher et al. [2014] identified such behavior for the GPU as well. However, the Android Power Profiler and V-edge do not take such hardware behavior into account. This is challenging, because the size of the workload and the operating condition define the state transitions. In the case of cellular networks, the operating condition also includes the network configuration (i.e., the number of states and also the inactivity timer settings for the corresponding states) [Hoque et al. 2014b]. For Wi-Fi, the value of the timer varies from 30 to 200ms [Hoque et al. 2014a]. DevScope tries to recognize such state transitions from the battery interface update samples. The RILAnalyzer application is an on-device tool for monitoring the RRC states of the 3G modem on some specific Android phones [Vallina-Rodriguez et al. 2013]. Although this application can be used by the profilers to identify 3G network configuration at runtime, it does not work with different chipsets and 4G networks. In addition, frequent polling of network information can create energy overhead. To this end, recognizing state transition from battery interface updates, as is done by DevScope, seems promising. Hardware also evolves continuously, and power consumption behavior becomes better and better understood. Therefore, new revised models, such as the wireless communication power models presented in Xiao et al. [2014], Ding et al. [2013], and Garcia-Saavedra et al. [2012], are required to be incorporated into the power models. This also applies for the display. Since different devices use different types of display [Hoque et al. 2013a], profilers should adopt improved models presented by Dong et al. [2009], Dong and Zhong [2012], and Chen et al. [2013].

10.2. Modeling Methodology

In the previous section, we discussed the impact of limited subcomponent coverage of power models on the performance of the profilers. We now discuss other kinds of limitations related to the modeling methodology. Recall that a white box model, which we also refer to as FSM, defines different power states and the triggers to transition from one state to another. Black box modeling uses statistics to fit a model to observations in the training phase. Usually, linear regression is used by the profilers, and they assume that power consumption always increases linearly with the resource usage regardless of the underlying power states. Therefore, in cases where the power consumption increases nonlinearly, the error increases. FSM-based profilers look into the power consumption in each power state. Given one power state, the power consumption may be static or may increase with the resource usage. Whether to choose linear regression, FSM, or a combination of them depends on the power consumption behavior of the subcomponents in question. Modeling components that exhibit tail energy are an example of a problematic case for simple linear regression. Nonlinear power consumption behavior can sometimes also be overcome by applying suitable transformations.

There are two main challenges pertaining to black box modeling. First, it requires expert knowledge about the features that are related to power consumption, which also holds for white box modeling. Including all possible features in a model is impractical, because collecting a lot of predictor or feature values creates overhead. In addition, some features may not be available, as a mobile OS typically exposes subcomponent information selectively. Therefore, it is important to find the most relevant features at runtime. Sesame does this by applying PCA [Dong and Zhong 2011]. Second, linear regression models distribute the weight of the coefficients across all coefficients when the features are correlated and reflected in the final model. Therefore, linear regression is appropriate when the features are independent. A piece-wise linear model can be used when the function behaves significantly different for different input sample values [Singh et al. 2009]. To reduce the effect of observation errors, several methods are used with linear regression models, such as total least square by Dong and Zhong [2011] and nonnegative least square by Diop et al. [2014].

Given the complexity of smartphones today, it is difficult to identify the relevant features or predictors. For example, regression with PCA in Dong and Zhong [2011] failed to identify Wi-Fi as one of the important contributors. Lasso regression [Hastie et al. 2001] can alleviate the dependency on domain-specific knowledge in determining the right features. In addition, it is less computationally complex and automatically selects a small set of relevant features. Linear and lasso models work remarkably well when the features are independent of each other. In the case of dependent features, linear models perform poorly, whereas nonlinear models can capture the dependencies often. For example, Bircher and John [2007] suggest that quadratic models are effective in power modeling. Another alternative approach is to use a support vector machine–based regression [McCullough et al. 2011], which handles the nonlinearity by mapping data into a higher-dimensional space. Consequently, the weight of the correlated features gets distributed.

10.3. Resource Utilization Sampling and Battery Update Rates

The profilers collect and feed the subcomponent utilization logs or predictor values to the models. Therefore, the rate at which the predictor values are collected has an impact on the time it takes to construct the model and the profiling accuracy. Whereas the update rate of the smart battery interface depends on the mobile vendor [Maker et al. 2013], the update rate of information related to other components depends on the OS and the application usage. For example, the highest reported update rate of smart battery interfaces is 4Hz, whereas a Linux OS updates the P-State residency of the CPU at a rate of 250Hz. Sesame improves the accuracy by averaging the collected samples. The more the averaging, the higher the accuracy but also the lower the update rate. For instance, Sesame yields an accuracy of 94% to 95% with a sampling rate below 1Hz, but the accuracy drops to 86% and 82% at 10Hz and 100Hz sampling rates, respectively. DevScope and PowerBooter, on the other hand, synchronize the smart battery update events with the component tests in the model generation phase, resulting in an accuracy of approximately 95%. In the case of V-edge, the error increases as the sampling delay increases beyond 3 seconds after the instantaneous voltage drops. In the case of event-based profilers, system call tracing also impacts accuracy through nonnegligible computation overhead, which increases the energy consumption. For instance, this overhead is reported to be 1% to 8% for Eprof.

10.4. Power Measurement

Power measurement is an integral part of the software power profilers. In Section 6, we discussed the on-device profilers that are independent of the off-device measurements and rely only on the smart battery interface to correlate the system power consumption with the battery depletion. Table IX illustrates that such measurements are error prone and thus require repetitive measurements; most of these profilers apply such repetitive methodology. The error is caused by imperfections of the battery models used. Although SOC-dependent profilers, such as PowerBooter, do not discuss the error originating from the underlying battery SOC-estimation mechanisms, such error should affect the accuracy of the profilers.

There are two sources of inaccuracy for the self-metering profilers. The first one is the SOC estimation error due to the battery model. The Rint model suffers from 11% SOC error [Shin et al. 2013]. The Thevenin and Coulomb counting approaches suffer from 5% [Chen and Rincon-Mora 2006] and 2.2% [Android 2014a] error, respectively. However, modern devices are being equipped with better fuel gauge ICs. Other than the simple Rint or Thevenin model, another approach exists that correlates the load voltage of the battery with the OCV voltage dynamically to calculate the SOC (e.g., the MAX17048 in the Nexus 5). In the case of Coulomb counting, accumulation of the current offset error can be compensated by charging/discharging the battery 100% and keeping the voltage stable for a while; therefore, an OCV-lookup table is required. Such a combined approach is used by the ModelGauge m3 enabled chip [Maxim 2014a] (e.g., the MAX17050 in the Nexus 6).

The second source of SOC error is the usable capacity estimation error [Hoque and Tarkoma 2015]. As the lithium-ion battery ages, the discharge rate of the battery

Table IX. Typical Measurement Methodologies for Component-Specific Power Consumption of a Mobile Device [Android 2014a]

Subcomponents	Measurement Methodology
Device power	Battery model–dependent measurement requires repeated measurements for a set of different configurations and then finds the difference in power consumption between the measurements. The battery model–specific errors are consistent across these measurements depending on the model used by the fuel gauge or the profiler.
Wi-Fi, GSM radio, Bluetooth activity, GPS	(1) Keep the device in the airplane mode when the target subcomponent does not have a wireless radio. (2) The wireless subcomponent should be in a specific mode (i.e., scanning, idle, transmit or receive), and the device should be free from the interference caused by other wireless sources, if required.
Display ON/OFF	(1) Keep the screen off when measuring the power consumption of other components. (2) To measure screen power consumption, the device should be in airplane mode, the brightness should be in a fixed level, and the screen may be completely black or white.
System suspend/ resume	(1) When the screen is off, the device may turn off some of the components or put them in a low power state. Therefore, the device must be kept in awaking mode or prevented from the suspension when measuring the power consumption of the desired component. (2) For measuring power consumption in suspend mode, the device should be in airplane mode and all wireless radios are disabled, and the device must be kept idle for a longer period of time so that power consumption becomes stable to a lower value.
CPU cores, frequency, and power states	Keep the number of CPU cores and their frequencies constant while carrying the CPU or other power measurements to avoid error.

increases for the same usage [Barré et al. 2014]. Consequently, the reported power consumption will be higher than the true value for an aged battery. For this reason, self-metering profilers may require retraining of their models. Hence, the accuracy of self-metering profilers is intertwined with the age of the battery and will decrease as the age of the battery increases, but none of the profilers addresses this issue at this moment. Precise quantification of its effect on the accuracy would require further evaluation of the profilers with batteries of different ages, and so far it is an open question for future research to address.

Some on-device and off-device profilers depend on power measurement with external tools like the Monsoon Power Monitor or BattOr [Schulman et al. 2011]. The accuracy of the profilers are also measured against the direct power measurement results. Therefore, the methodology followed during the measurement plays an important role in the accuracy of the measurement and thus the accuracy of the software profilers. One can measure the power consumption of a component by comparing the power consumption at the desired state (e.g., the power consumption of Wi-Fi in the idle state). If premeasures are not taken, then the other external influencing factors, such as interference or other broadcasts, may bias the Wi-Fi measurement result [Peng et al. 2015]. Again, if the smartphone is plugged into a computer or a wall charger during power measurements, a measurement error is possible because current may flow into the device. For instance, in the measurement setup presented by Banerjee et al. [2014], the smartphone was connected to a desktop computer. Table IX lists the component-specific guidelines that would help to produce power measurements with higher accuracy.

11. USABILITY

In general, profilers enable application developers to understand the performance of their program code and to tune their applications to reduce the energy consumption. Recent studies suggest that users are also concerned about the power consumption of their mobile devices [Pathak et al. 2011; Jung et al. 2014], and their energy awareness can improve the performance of their devices [Athukorala et al. 2014]. Therefore, the profilers also guide the users to point out the most energy-consuming applications running in their devices. However, the choice of a profiler for a typical user or developer is not straightforward. Along with their accuracy (see Table VII), other important factors are their availability, requirements of the user, hardware support, ease of use and installation, and expertise of the user. Overall, it is fair to say that most profilers provide an acceptable accuracy for most use cases, which suggests that other features, such as software availability and usability, subcomponent coverage, and profiling rate, may weigh more than accuracy when choosing which profiler to use.

Table X describes the ease of installation and usage of the profiler applications, information offered by the profilers, and the way they provide the visualization of the measurement results. We can see that most of the profilers work on Android devices. Although they are available as mobile applications, their usability is limited by at least one or more factors. For example, Trepn works better in devices with MDP and specific fuel gauge ICs, PowerTutor requires rooting of the device for estimating display power consumption more accurately, and AppScope requires instrumenting the kernel and hence rooting the device. Consequently, they are suitable for researchers or application developers and remain difficult to use for average consumers. Usability may also be limited by the dependency on the underlying fuel gauge chip used by the mobile devices. For example, the profilers depending on current readings from the battery interface will not work with devices that only provide voltage readings.

On-device power profilers are interesting for all kinds of users, consumers, application developers, and researchers, whereas offline profilers are usually of interest only to developers and researchers. Furthermore, off-device modeling that requires instrumenting the smartphone with an external power monitor is not possible even for most application developers, not to mention consumers. Another aspect to consider is that profilers with on-device modeling will provide consistent accuracy with different devices of the same model, different device models, and usage scenarios. In contrast, profilers with off-device models are device specific and require measurements and model calibration for every device. The off-device models in Android devices are calibrated by the manufacturer. They either conduct measurements or collect energy ratings of the individual subcomponents from system-on-chip manufacturers, which is close to impossible for an individual researcher to conduct in a laboratory. In addition, it is not always easy to instrument the devices for power measurements.

12. CONCLUSIONS

This article provides a broad survey of smartphone energy profilers. We paid special attention to their accuracy reported by the authors and to the way it has been evaluated, to the profiling coverage in terms of components considered, to power modeling methodology used, and their usability. The accuracy of the surveyed profilers varies between 86% and 97.7%, and for some profilers the accuracy could be increased to greater than 90% by modeling the power consumption of the relevant subcomponents and synchronizing the sampling frequency with smart battery interface updates. Although on-device models are inferior to off-device models accuracy wise, the difference is by no means dramatic.

We found a rather large variation in terms of profiling coverage. Although most profilers do not include power models for the GPU, the range of wireless network interfaces modeled differs a lot between the profilers. Concerning modeling methodology, we observed that only Sesame makes an attempt to automatically select the best predictors. We believe that there is room to improve the power modeling methodology, at

Table X. Usability and Measurement Information Offered by Profilers and Debuggers

Profiler	Usability and Visualization	Offered Information	Target User
NEP	On-device stand-alone profiling. Easy installation. Graph windows are easily accessible. It works only in Symbian devices.	Total system power consumption in watts or amps, HSPA timers, data transmit and receive, wireless signal strength, and discharge curve. All of this information is presented in temporal graphs and can be exported to a desktop computer for further analysis.	Expert user
Trepn	On-device stand-alone profiling. Easy installation. The UI displays temporal graphs that are easily accessible, and the plots can be seen on the foreground of a running application to see the power consumption and resources used by the application. The profiler works only on Snapdragon chipset-based Android devices powered with special component-wise sense resistors and power management ICs.	Component-specific power consumption and utilization, and the information can be exported to a desktop computer for offline analysis.	Expert user
Power Tutor	Easy installation but requires rooting of the Android devices for more accurate power measurement of the display. The application UI depicts easy visualization of component power consumption as line graphs and the share of individual components in total power consumption as a pie chart. Requires device-specific calibration.	Component- and application-specific power consumption. The measurement can be exported for further analysis.	Expert user
AppScope	Requires rooting of the device and device-specific calibration.	Measurement data can be exported for further analysis.	Expert user
Android Power Profiler	Default Android system power profiler. The application UI shows the percentage of energy used by the display and other applications since the last time the device was powered on. It also displays other related information, such as signal strength and the number of charging events.	Display- and application-specific power consumption.	Average user
Carat	Easy installation. The UI presents the performance of a device across the community of similar devices, the visualization of the energy-buggy and energy-hoggy applications running in the device, and the improvement in terms of battery life if the user kills a hoggy application.	Lists of energy-buggy and energy-hoggy applications, energy benefit, and the explanation of these terms.	Average user

least in the case of black box modeling. Factors related to usability differ substantially between profilers. Whereas some profilers could easily be used by consumers, others require such expert knowledge that only trained professionals could take advantage of them. In addition, the availability of profiler software varies, with some being openly available, whereas most are nondisclosed research prototypes. In addition, all profilers are OS specific, and some work only on certain device models.

ACKNOWLEDGMENTS

We are grateful to Professor Jon Crowcroft from the University of Cambridge for his valuable feedback during the preparation of the survey. We also thank Dr. Eemil Lagerspetz for advice and insights pertaining to the Carat system.

REFERENCES

Android. 2014a. Android Power Profiles. Retrieved February 20, 2014, from https://source.android.com/devices/tech/power.html.

Android. 2014b. Battery Historian. Retrieved January 7, 2015, from https://developer.android.com/about/versions/android-5.0.html.

Android. 2014c. Hierarchy Viewer: Debugging and Profiling UIs. Retrieved November 20, 2014, from http://developer.android.com/tools/help/hierarchy-viewer.html.

Android. 2015. Android WifiManager.WifiLock. Retrieved April 30, 2015, from http://developer.android.com/reference/android/net/wifi/WifiManager.WifiLock.html.

Kumaripaba Athukorala, Eemil Lagerspetz, Maria von Kügelgen, Antti Jylhä, Adam J. Oliner, Sasu Tarkoma, and Giulio Jacucci. 2014. How Carat affects user behavior: Implications for mobile battery awareness applications. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems (CHI'14)*. ACM, New York, NY, 1029–1038.

Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. 2009. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference (IMC'09)*. ACM, New York, NY, 280–293.

Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 588–598.

Anthony Barré, Frédéric Suard, Mathias Gérard, and Delphine Riu. 2014. A real-time data-driven method for battery health prognostics in electric vehicle use. In *Proceedings of the 2nd European Conference of the Prognostics and Health Management Society 2014 (PHMCE'14)*. 1–8.

Frank Bellosa. 2000. The benefits of event: Driven energy accounting in power-sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop: Beyond the PC—New Challenges for the Operating System*. ACM, New York, NY, 37–42.

W. Lloyd Bircher and Lizy K. John. 2007. Complete system power estimation: A trickle-down approach based on performance events. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'07)*. IEEE, Los Alamitos, CA, 158–168.

Niels Brouwers, Marco Zuniga, and Koen Langendoen. 2014. NEAT: A novel energy analysis toolkit for free-roaming smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys'14)*. ACM, New York, NY, 16–30.

Martin Burtscher, Ivan Zecena, and Ziliang Zong. 2014. Measuring GPU power with the K20 built-in sensor. In *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU-7)*. ACM, New York, NY, Article No. 28.

Min Chen and Gabriel A. Rincon-Mora. 2006. Accurate electrical battery model capable of predicting runtime and I-V performance. *IEEE Transactions on Energy Conversion* 21, 2, 504–511.

Xiang Chen, Yiran Chen, Zhan Ma, and Felix C. A. Fernandes. 2013. How is energy consumed in smartphone display applications? In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications (HotMobile'13)*. ACM, New York, NY, Article No. 3.

Xiang Chen, Jian Zheng, Yiran Chen, Mengying Zhao, and Chun Jason Xue. 2012. Quality-retaining OLED dynamic voltage scaling for video streaming applications on mobile devices. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. ACM, New York, NY, 1000–1005.

Gilberto Contreras and Margaret Martonosi. 2005. Power prediction for Intel XScale processors using performance monitoring unit events. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'05)*. ACM, New York, NY, 221–226.

Gerard Bosch Creus and Mika Kuulusa. 2007. Optimizing mobile software with built-in power profiling. In *Mobile Phone Programming*, Frank H. P. Fitzek and Frank Reichert (Eds.). Springer, Netherlands, 449–462.

Eduardo Cuervo, Aruna Balasubramanian, Dae-Ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*. ACM, New York, NY, 49–62.

Ning Ding, Daniel Wagner, Xiaomeng Chen, Abhinav Pathak, Y. Charlie Hu, and Andrew Rice. 2013. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'13)*. ACM, New York, NY, 29–40.

Tahir Diop, Natalie Enright Jerger, and Jason Anderson. 2014. Power modeling for heterogeneous processors. In *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU-7)*. ACM, New York, NY, Article No. 90.

Mian Dong, Yung-Seok Kevin Choi, and Lin Zhong. 2009. Power modeling of graphical user interfaces on OLED displays. In *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*. ACM, New York, NY, 652–657.

Mian Dong and Lin Zhong. 2011. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, 335–348.

Mian Dong and Lin Zhong. 2012. Power modeling and optimization for OLED displays. *IEEE Transactions on Mobile Computing* 11, 9, 1587–1599.

Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. 2010. Diversity in smartphone usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*. ACM, New York, NY, 179–194.

Jason Flinn and Mahadev Satyanarayanan. 1999. PowerScope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*. IEEE, Los Alamitos, CA.

Andres Garcia-Saavedra, Pablo Serrano, Albert Banchs, and Giuseppe Bianchi. 2012. Energy consumption anatomy of 802.11 devices and its implication on modeling and design. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT'12)*. ACM, New York, NY, 169–180.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. *The Elements of Statistical Learning*. Springer, New York, NY.

Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 280–289.

Mohammad A. Hoque, M. Siekkinen, and Jukka K. Nurminen. 2014a. Energy efficient multimedia streaming to mobile devices: A survey. *IEEE Communications Surveys Tutorials* 16, 1, 579–597.

Mohammad A. Hoque, M. Siekkinen, Jukka K. Nurminen, and M. Aalto. 2013a. Dissecting mobile video services: An energy consumption perspective. In *Proceedings of the 2013 IEEE 14th International Symposium and Workshops on a World of Wireless, Mobile, and Multimedia Networks (WoWMoM'13),* IEEE, Los Alamitos, CA, 1–11.

Mohammad A. Hoque, Matti Siekkinen, and Jukka K. Nurminen. 2013b. Using crowd-sourced viewing statistics to save energy in wireless video streaming. In *Proceedings of the 19th Annual International Conference on Mobile Computing and Networking*. ACM, New York, NY, 377–388.

Mohammad A. Hoque, Matti Siekkinen, Jukka K. Nurminen, Sasu Tarkoma, and Mika Aalto. 2014b. Saving energy in mobile devices for on-demand multimedia streaming—a cross-layer approach. *ACM Transactions on Multimedia Computing, Communications, and Applications* 10, 3, Article No. 25.

Mohammad A. Hoque and Sasu Tarkoma. 2015. Sudden drop in the battery level? Understanding smartphone state of charge anomaly. In *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower'15)*. ACM, New York, NY, 26–30.

Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. A close examination of performance and power characteristics of 4G LTE networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12)*. ACM, New York, NY, 225–238.

Anders R. Jensen, Mads Lauridsen, Preben Mogensen, Troels B. Srensen, and Per Jensen. 2012. LTE UE power consumption model: For system level energy and performance optimization. In *Proceedings of the 2012 IEEE Vehicular Technology Conference (VTC Fall'12).* IEEE, Los Alamitos, CA, 1–5.

Wonwoo Jung, Yohan Chon, Dongwon Kim, and Hojung Cha. 2014. Powerlet: An active battery interface for smartphones. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'14)*. ACM, New York, NY, 45–56.

Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. 2012. DevScope: A nonintrusive and online power analysis tool for smartphone hardware components. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, New York, NY, 353–362.

Jim Keniston, Prasanna S. Panchamukhi, and Masami Hiramatsu. 2011. *Kernel Probes*. Technical Report. Retrieved October 27, 2014, from https://www.kernel.org/doc/Documentation/kprobes.txt.

Eamonn Keogh, Jessica Lin, and Ada Fu. 2005. HOT SAX: Efficiently finding the most unusual time series subsequence. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM'05)*. IEEE, Los Alamitos, CA, 226–233.

Mikkel Baun Kjaergaard and Henrik Blunck. 2012. Unsupervised power profiling for mobile devices. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Lecture Notes of the Institute for Computer Sciences, Social Informatics, and Telecommunications Engineering, Vol. 104. Springer, 138–149.

Mads Lauridsen, Preben Mogensen, and Laurent Noel. 2013. Empirical LTE smartphone power model with DRX operation for system level simulations. In *Proceedings of the 2013 IEEE 78th Vehicular Technology Conference (VTC Fall'13)*. 1–6.

Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. *SIGARCH Computer Architecture News* 41, 3, 487–498.

Tao Li and Lizy Kurian John. 2003. Run-time modeling and estimation of operating system power consumption. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*. ACM, New York, NY, 160–171.

Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. 2009. Statistical power consumption analysis and modeling for GPU-based computing. In *Proceedings of the ACM SOSP Workshop on Power Aware Computing and Systems (HotPower'09)*. ACM, New York, NY.

Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. 2013. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. 57–70.

Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, New York, NY, 224–234.

Frank Maker, Rajeevan Amirtharajah, and Venkatesh Akella. 2013. Update rate tradeoffs for improving online power modeling in smartphones. In *Proceedings of the 2013 IEEE International Symposium on Low Power Electronics and Design (ISLPED'13)*. IEEE, Los Alamitos, CA, 114–119.

Maxim. 2014a. *MAX17047/MAX17050, ModelGauge m3 Fuel Gauge*. Technical Report. Retrieved November 9, 2015, from http://datasheets.maximintegrated.com/en/ds/MAX17047-MAX17050.pdf.

Maxim. 2014b. *MAX17048/MAX17049, Micropower 1-Cell/2-Cell Li+ ModelGauge ICs*. Technical Report. Retrieved November 9, 2015, from http://datasheets.maximintegrated.com/en/ds/MAX17048-MAX17049.pdf.

John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. 2011. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIXATC'11)*. 12.

Microsoft. 2010. Nokia Energy Profiler. Retrieved December 15, 2014, from http://developer.nokia.com/community/discussion/showthread.php/160912-Nokia-Energy-Profiler/page8.

Monsoon. 2014. Power Monitor. Retrieved November 9, 2015, from https://www.msoon.com/LabEquipment/PowerMonitor/.

Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. 2010. Statistical power modeling of GPU kernels using performance counters. In *Proceedings of the International Conference on Green Computing (GREENCOMP'10)*. IEEE, Los Alamitoa, CA, 115–122.

Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. 2013. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, Article No. 10.

OpenBinder. 2005. Binder Overview. Retrieved November 9, 2015, from http://www.angryredplanet.com/hackbod/openbinder/docs/html/BinderOverview.html.

Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 2011. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks (HotNets-X)*. ACM, New York, NY, Article No. 5.

Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 2012. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, New York, NY, 29–42.

Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. 2011. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the 6th Conference on Computer Systems*. ACM, New York, NY, 153–168.

Ge Peng, Gang Zhou, David T. Nguyen, and Xin Qi. 2015. All or none? The dilemma of handling WiFi broadcast traffic in smartphone suspend mode. In *Proceedings of the 2015 IEEE INFOCOM (INFOCOM'15)*. IEEE, Los Alamitos, CA, 9.

PowerTutor. 2009. PowerTutor: A Power Monitor for Android-Based Mobile Platforms. Retrieved November 9, 2015, from http://ziyang.eecs.umich.edu/projects/powertutor/documentation.html.

Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. 2011. Profiling resource usage for mobile applications: A cross-layer approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*. ACM, New York, NY, 321–334.

Qualcomm. 2014a. MDP Power Rail. (2014). Retrieved November 15, 2014, from https://developer.qualcomm.com/forum/qdn-forums/increase-app-performance/trepn-profiler/27700.

Qualcomm. 2014b. Trepn Profiler. Retrieved June 11, 2014, from https://developer.qualcomm.com/trepn-profiler.

Swaminathan Vasanth Rajaraman, Matti Siekkinen, and Mohammad Hoque. 2014. Energy consumption anatomy of live video streaming from a smartphone. In *Proceedings of the 25th IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications. (PIMRC'14)*. IEEE, Los Alamitos, CA.

Santhosh Kumar Rethinagiri, Oscar Palomar, Rabie Ben Atitallah, Smail Niar, Osman Unsal, and Adrian Cristal Kestelman. 2014. System-level power estimation tool for embedded processor based platforms. In *Proceedings of the 6th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'14)*. ACM, New York, NY, Article No. 5.

Seyed Mohammad Rezvanizaniani, Zongchang Liu, Yan Chen, and Jay Lee. 2014. Review and recent advances in battery health monitoring and prognostics technologies for electric vehicle (EV) safety and mobility. *Journal of Power Sources* 256, 110–124.

Thorsten Schreiber. 2011. Android Binder: Android Interprocess Communication. Retrieved November 9, 2015, from https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf.

Aaron Schulman, Thomas Schmid, Prabal Dutta, and Neil Spring. 2011. Demo: Phone Power Monitoring with BattOr. Retrieved November 9, 2015, from http://www.web.stanford.edu/∼aschulm/docs/mobicom11-phone-powermonitor-demo.pdf.

Donghwa Shin, Kitae Kim, Naehyuck Chang, Woojoo Lee, Yanzhi Wang, Qing Xie, and Massoud Pedram. 2013. Online estimation of the remaining energy capacity in mobile systems considering system-wide power consumption and battery characteristics. In *Proceedings of the 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC'13)*. IEEE, Los Alamitos, CA, 59–64.

Alex Shye, Benjamin Scholbrock, and Gokhan Memik. 2009. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, 168–178.

Matti Siekkinen, Mohammad A. Hoque, Jukka K. Nurminen, and Mika Aalto. 2013. Streaming over 3G and LTE: How to save smartphone energy in radio access network-friendly way. In *Proceedings of the 5th ACM Workshop on Mobile Video (MoVid'13)*. ACM, New York, NY.

Karan Singh, Major Bhadauria, and Sally A. McKee. 2009. Real time power estimation and thread scheduling via performance counters. *SIGARCH Computer Architecture News* 37, 2, 46–55.

Lindsay I. Smith. 2002. A Tutorial on Principle Components Analysis. Retrieved November 9, 2015, from http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf.

Texas Instruments. 1999. *Understanding Data Converters*. Technical Report. Retrieved November 9, 2015, from http://www.ti.com/lit/an/slaa013/slaa013.pdf.

Kuen Hung Tsoi and Wayne Luk. 2011. Power profiling and optimization for heterogeneous multi-core systems. *SIGARCH Computer Architecture News* 39, 4, 8–13.

Bogdan Marius Tudor and Yong Meng Teo. 2013. On understanding the energy consumption of arm-based multicore servers. *SIGMETRICS Performance Evaluation Review* 41, 1, 267–278.

Narseo Vallina-Rodriguez, Andrius Auçinas, Mario Almeida, Yan Grunenberger, Konstantina Papagiannaki, and Jon Crowcroft. 2013. RILAnalyzer: A comprehensive 3G monitor on your phone. In *Proceedings of the 2013 Internet Measurement Conference (IMC'13)*. ACM, New York, NY, 257–264.

Narseo Vallina-Rodriguez and Jon Crowcroft. 2013. Energy management techniques in modern mobile handsets. *IEEE Communications Surveys Tutorials* 15, 1, 179–198.

Yu Xiao, Rijubrata Bhaumik, Zhirong Yang, Matti Siekkinen, Petri Savolainen, and Antti Yla-Jaaski. 2010. A system-level model for runtime power estimation on mobile devices. In *Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical, and Social Computing (GREENCOM-CPSCOM'10)*. IEEE, Los Alamitos, CA, 27–34.

Yu Xiao, Yong Cui, Petri Savolainen, Matti Siekkinen, An Wang, Liu Yang, Antti Yla-Jaaski, and Sasu Tarkoma. 2014. Modeling energy consumption of data transmission over Wi-Fi. *IEEE Transactions on Mobile Computing* 13, 8, 1760–1773.

Fengyuan Xu, Yunxin Liu, Qun Li, and Yongguang Zhang. 2013. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. 43–56.

Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. 2012. AppScope: Application energy metering framework for Android smartphones using kernel activity monitoring. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*. 36.

Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. 2010. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, New York, NY, 105–114.

Yifan Zhang, Xudong Wang, Xuanzhe Liu, Yunxin Liu, Li Zhuang, and Feng Zhao. 2013. Towards better CPU power management on multicore smartphones. In *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower'13)*. ACM, New York, NY, Article No. 11.

Received February 2015; revised July 2015; accepted October 2015