# Predicting Power Usage of Android Applications

Benjamin Bramble
UW-Madison

Michael Swift
UW-Madison

May 2014

**Abstract**

*Android based devices have become increasing important in many peoples' lives. The increasing number of device components like cameras, WIFI, and multicore processors as well as increasingly complex applications drain the battery leading to frustrated users who depend on non-stop access to their device. We propose a solution named SApp that will educate the user on the impact of their applications by providing an opportunity to reduce the battery consumption through smarter user decisions and prioritization. This is a three step process: test the impact of the specific device components, then measure the performance of applications, and combine heuristics with the measurements to predict future application impacts. The end result is an educated user capable of extending the life of their device through informed decisions.*

## 1 Introduction

The Android Operating System is the most popular in the world, powering over one billion mobile devices[5]. These devices provide users with the ability to communicate, play games, and browse the internet anywhere at any time. The popularity has led to manufacturers adding more processors for increasing power, stronger network transmitters for faster data sharing, and larger, clearer screens for better viewing experiences. The changes improve the enjoyment of the device, but weight and slow improvements in battery technology limit the energy a device can store. In an online poll of over 7,000 Android users, 72% said that they were dissatisfied with the battery life of their phone[4].

The initial assumption would be to improve the hardware for a more power efficient display or longer lasting batteries. However, there exists a popular market[8][9][7][12][14] for a software approach that informs users and developers about the cause of their device's power drain and the attributing applications. Developers could use that information to design more energy efficient programs while users could limit the usage of power hunger application to lengthen battery life.

We present SApp as a potential solution to educate users and developers alike by taking a three step approach to application energy profiling. The first step builds a power model of popular device components such as the CPU, display, IO, and WIFI. The model is built using a software-only approach that is unique to the specific device at a specific point in

time. With this capability, as the hardware degrades over time, the model can be rebuilt to reflect the current operating environment.

The second step runs as a background service and collects statistics relating specific applications to the various components. SApp's background service is designed to produce low energy overhead while accurately assigning cost to processes. The service runs at the kernel level by adding statistical counters to the Linux process structure to record CPU by frequency, IO reads and writes, and bytes received over WIFI and outputs that data into the processes /proc folder. The service also differentiates the foreground applications from the background applications to ensure proper accounting of the display.

The final step aggregates the data collected in step two and combines it with the power model obtained in step one to accurately reflect the allocation of power. Furthermore, that information can be used to predict future battery drain to provide users an understanding of which applications they can use for a certain duration while maintaining above a battery threshold. The end state is an informed user or developer capable of extending the battery life of their device through informed, conscience decisions.

## 2 Motivation

The increasing complexity of mobile devices and the user tendency to multitask applications such as games, email, music, and video players has led to situations where the battery life seemingly disappears. At least that is how it appears to the user. In actuality, the running applications, whether in the foreground or background, are accessing the various phone components such as the CPU, WIFI, display, or conducting IO. In order to make power efficient decisions, the user must understand where the power drain comes from and the battery cost as attributed to specific applications on their specific device.

For this project, we aimed to create an application that can determine the cost of all running applications by collecting the component use by application and outputting the expected battery drain of continued use. The ideal situation is that of an individual waiting to board an airplane. That person has limited power left on their cell phone with no opportunity to recharge prior to landing. However, the individual desires to play a game for twenty minutes prior to boarding, but needs to have enough power left to make a phone call following the flight. Based on prior phone use, SApp will predict the energy required to place and phone call and calculate the duration the game can be played prior to reaching the phone call energy threshold. SApp will inform the user that they can play ten minutes to preserve enough battery for a phone call.

What separates SApp from current power models is a software only approach that creates a unique profile to the specified device at that specific time. In addition, SApp takes the power model to the next step by offering predictive drain based on previous performance. The intent is to end with a more informed user, capable of making energy-aware decisions to extend the battery life between charges. In addition, SApp benefits application developers by allowing them to profile the component usage of their design decisions and reduce energy cost whenever possible. To add additional focus, research indicates that graphics, CPU, and networking drain the most power from mobile devices [11].

# 3 Design

To achieve the aforementioned goals, SApp is broken down into three steps. First, SApp determines a measurable cost of using a specific device component for a length of time. This step includes building a power model for the device by isolating components and determining the battery drain over a unit of time. The second step runs as a background service, collecting application data. The information collected is broken down to component statistics that are correlated to the model developed in step one. The final step aggregates the data collected in step two and applies the cost coefficients built in step one to the data collected in step two to determine the cost of an application. That information is presented to the user and used to predict battery drain for future application usage. Ultimately, as discussed below, SApp requires a rooted device with a customized kernel. This design decision is unavoidable based on the requirements of the test suite and application polling. Admittedly, both of these decisions negatively impact the market for SApp.

## 3.1 Profiling

To build a power model of a given device, SApp runs a testing suite aimed at isolating specific components to determine their cost to the system of a whole. This test is designed to run over the course of several hours without other applications sharing resources, ideally overnight. Currently, the test is composed of four component modes: display, CPU, IO, and WIFI. Figure 1 displays the specific subtests that isolate the various components. Certainly, more components should be added for system completeness but went beyond the timeline and scope for this project.

For consistency during the tests, the CPU is held constant with a steady workload on all tests other than testing CPU frequencies. Android devices typically operate with the OnDemand governor which dynamically scales the CPU frequency based on CPU utilization[6]. For control purposes, every test mode outside of CPU sets all CPU's to the Userspace governor and sets the maximum and minimum CPU frequency to the median frequency option. In addition, each test mode creates a spin thread for each processor to occupy all cores and eliminate CPU idle time which drains less power.

Each test mode determines when the battery changes one full percentage which is the most accurate power measurement provided by Android. SApp captures the CPU time by frequency in cycles, IO statistics in bytes read and written to disk, WIFI bytes received, and total time elapsed in milliseconds. We chose WIFI bytes received as the initial networking representative due to the nature of mobile devices to receive more than transmit for applications such as the browser or games. Each test mode runs for three iterations, with one iteration equaling one percent battery drain, then keeps the test iteration with the median runtime. Repeated tests showed that test runtimes were consistent with some fluctuation due to background processes. When possible, components such as WIFI are turned off until testing to provide further isolation.

The first test examines the display by turning off the WIFI, running one spin thread per CPU, setting the CPU frequencies to a median value, and maintaining a pure white display at 70% brightness. The screen power setting most resembles that of most applications and the white background was chosen as it is the most expensive color per pixel[11]. As I will

discuss in the future work section, the display test could be improved significantly with a more inclusive drill down into the GPU. However, due to time constraints, the current test fills nicely as a broader stand-in.

After three iterations of display testing, the screen lock is released and allowed to turn off. The CPU test mode captures all possible CPU frequencies and randomizes the order in which they are tested. Each frequency option is tested for three iterations with one spin thread for each processor and the WIFI turned off.

The third test analyzes the effect of IO on battery drain by keeping the screen off, setting all CPU frequencies to the median option, maintaining a spin thread per CPU, and conducting tests that perform specific IO functions. Two IO tests are conducted: one for reads and another for writes. Both tests conduct 1,200 operations of 32,000 bytes capacity on both internal storage and an SD card. The IO operations read or write large, random-filled files from random offsets to force cache misses. SApp tested a variety of different tests altering the number and size of reads before settling on the current form as a decent indicator of power drain. The number of bytes actually coming from or reaching disk are read from /proc/[PID]/IO.

The WIFI test maintains the same frequencies, spin cycles, and screen settings as the IO test. At this point, the WIFI is activated and connected to a local preset WIFI network. SApp makes repeated requests from www.android.com and records the number of bytes received.

Following the completion of all tests, the data is converted into component cost coefficients. Only the median test iteration results are examined with all other iterations discarded. First, the CPU energy coefficients are calculated by dividing 1% battery by the number of CPU cycles at that frequency as indicated in formula 1.

$$E_f = \frac{1}{CyclesAtFreq_f} \tag{1}$$

Second, the display energy coefficient is calculated by subtracting the CPU cost from 1% battery and dividing the remaining power drain by the time in milliseconds that the display was on as indicated in formula 2.

$$E_d = \frac{1 - E_f * CyclesAtFreq_f}{time} \tag{2}$$

Third, the IO energy coefficients are calculated by subtracting the CPU cost from 1% battery and dividing the remaining power drain by the number of bytes either read to or written from disk as indicated in formula 3.

$$E_r = \frac{1 - E_f * CyclesAtFreq_f}{bytes_read} \qquad E_w = \frac{1 - E_f * CyclesAtFreq_f}{bytes_written} \tag{3}$$

Finally, the WIFI energy coefficient is calculated by subtracting the CPU cost from 1% battery and dividing the remaining power drain by the number of bytes received as indicated in formula 4.

$$E_w = \frac{1 - E_f * CyclesatFreq_f}{bytes_received} \tag{4}$$

The results are then stored for use in assigning past cost per application and predicting future cost.

| Components with Test Modes | | | |
|---|---|---|---|
| Display | CPU (freqs) | IO (Op:Size of IO Ops:# of IO Ops) | WIFI |
| Display On | 384000 | Read:32000:1200 | Receive |
| | 486000 | Write:32000:1200 | |
| | 594000 | | |
| | 702000 | | |
| | 810000 | | |
| | 918000 | | |
| | 1026000 | | |
| | 1134000 | | |
| | 1242000 | | |
| | 1350000 | | |
| | 1458000 | | |
| | 1512000 | | |

Figure 1: Different Testing Modes for the Component Power Model

We make the basic assumption that only limited background applications are running. The user is expected to turn off any additional services or applications to ensure that SApp isolates resources. In a modern OS, it is nearly impossible to expect that all background threads such as the garbage collector or application manager. Some resource sharing during the component tests are acceptable as long as major applications are not run concurrently. Idealy, SApp is the only application running during the tests without interruption.

### 3.2 Application Polling

To assign cost to specific applications, SApp polls the Android Activity Manager for a list of every running application. Every thirty seconds, component usage statistics for each running application are recorded from each processes /proc/[PID] folder. The only problem is that the Android OS does not maintain many important statistics that SApp needs to access. For instance, the /proc/[PID] folder maintains IO statistics, but requires a rooted phone to read them. In addition, while user CPU time and system CPU time are maintained by the system, Android does not track the CPU time per frequency per application. As indicated in previous studies [17], the CPU frequency affects the rate at which energy is drained. SApp could have polled the current frequency and assumed that it was maintained for thirty seconds or more but that would most likely result in low accuracy. Lastly, while the bytes received and transmitted are recorded by networking interface, they are never attributed to a specific process. For these reason, we found that the Android Linux kernel does not maintain sufficient statistics for our power model.

The solution, and major limitation of SApp, came from modifying the kernel to record CPU frequency time, network bytes received, and IO bytes read from and written to disk in a single location in each process's /proc/[PID] folder. The single location reduces the number of reads and writes attributed to SApp lowering the IO cost of the poller. The new

kernel provides an accurate accounting of each process's resource use accurately and with low overhead.

The background service operates by waking every thirty seconds to grab a statistical snapshot of all currently running applications. After thirty more seconds, the delta of the statistics for all running processes are calculated and inserted into a SQLite database for use in assigning and projecting cost. It is assumed that very short running applications might not be captured in the thirty second window, but short running applications are less likely to have a major effect on the battery drain. SApp maintains two states of applications: foreground and background. Foreground applications reflect applications that are actively using the display and receiving user input. However, even when closed, applications continue to use resources so they must be tracked separately to avoid affecting the cost analysis of running applications. As the results show, users and developers may be interested in how much battery an application drains when relegated to the background.

### 3.3 Projecting

Upon termination of the application profiler, whether through a timer or user input, a background service aggregates the data collected. The service summarizes all statistical categories for each application while differentiating between when an application runs in the foreground and background. Next, the service multiplies the aggregated statistics for each application by the component coefficient determined in step one to find the cost of each application separated by CPU, display, IO, and WIFI. That cost is divided by the total time the application ran to determine the cost of the application per second. That simple coefficient can be used to provide the user with a cost estimate per length of time.

## 4 Results

The goal was to give the user a reasonable analysis of the cost of running various applications, not to provide a completely accurate accounting of the energy drain of a device. All output is given in battery percentage of the total system. Although less scientific as compared to watts or ampere-hour, many users of other application power diagnostic tools expressed an easier to understand output[14]. All tests were performed on a 2013 Nexus 7 with a customized Android 4.4.2. The device features a quad-core 1.5GHz processor.

### 4.1 Power Model

As indicated in Figure 2, the display drains battery life the fastest. As a result of the test not taking into account GPU calls, different pixels on the screen, or user inputs such as touches, all applications displayed in the foreground for the same duration will have the same display cost. SApp's findings reflect the importance of the display with those of other power models.

For this architecture, CPU cycles are measured in 1/100's of a second allowing for 10 seconds of CPU time to be compared to 10 seconds of display time in Figure 2 The CPU power cost reflects a linear relationship between CPU frequency and power drained. This also comes as no surprise as it is articulated in other power models as well. The results also

confirm the importance of a governor to scale back the frequency as the CPU utilization falls in order to maintain power efficiency.

The cost of the IO was the most unstable with some difficulty correlating the statistics gathered by the kernel with the drain of battery. Ultimately, IO accounted for a very small percentage of battery decrease so a conservative coefficient relating bytes read from and written to disk was determined. As indicated in Figure 2, ten seconds of display is still larger than 1MB IO operations signifying the lack of an IO power drain. As most application do not read or write very much data to disk, we are comfortable with the decision. The WIFI coefficient indicates that bytes transmitted over the network are more costly than IO operation although I only record bytes received and not bytes transmitted.

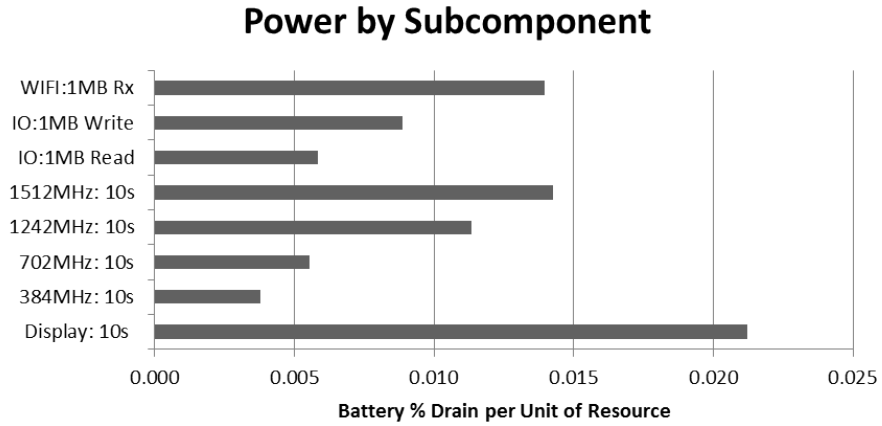


Figure 2: Test Results from a Component Test used to build a Power Model

### 4.2 Application Data Collection

The data collected during application polling shows some interesting trends. To start, after an application is minimized, it often continues in the background to reduce the time needed to bring it back to the forefront for continued use. While this feature is beneficial if one wants to use the application again shortly, it can also lead to wasted resources. As shown in Figure 3, Angry Birds in particular used thousands of CPU cycles at a relatively high frequency just to maintain state in the background. The Android browser also continued downloading data from the WIFI network, most likely for future buffering. Since we never restarted the application during that test run, it resulted in wasted battery drain. As a takeaway, users should terminate programs that they do not wish to use again in the near future and application developers should be aware of their energy drain during maintenance states.

The other important takeaway from Figure 3 is the relatively low energy cost of SApp. While write heavy due to the database storage, all other collected resource usage is low. SApp could consolidate and compact writes to further reduce the costs. One could observe the usefulness of this energy profiler in developing more energy efficient programs.

| | | | Frequency (cycle) | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | time(ms) | display(ms) | 384000 | 486000 | 594000 | 702000 | 810000 | 918000 | 1026000 | 1134000 | 1242000 | 1350000 | 1458000 | 1512000 | read(KB) | write(KB) | rx(KB) |
| System Services | 3786050 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 365 | 845 | 0 | 0 | 0 | 7985 | 0 | 104 | 10373 |
| Angry Birds Foreground | 2168274 | 2168274 | 0 | 0 | 0 | 0 | 0 | 0 | 6917 | 36232 | 0 | 0 | 0 | 229593 | 237568 | 4356 | 4243 |
| Angry Birds Background | 1617776 | 0 | 0 | 29 | 35 | 30 | 24 | 42 | 18218 | 2428 | 0 | 0 | 0 | 20753 | 0 | 64 | 9 |
| Browser Background | 2199242 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 99 | 179 | 0 | 0 | 0 | 2699 | 401408 | 6208 | 2335 |
| Browser Foreground | 1586808 | 1586808 | 0 | 0 | 0 | 0 | 1 | 1 | 387 | 454 | 0 | 0 | 0 | 8411 | 16384 | 9816 | 1041 |
| Power Profiler | 3786050 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 151 | 108 | 0 | 0 | 0 | 3890 | 0 | 9004 | 5 |

Figure 3: Aggregated Sample Results from a one hour Application Polling Run with all Output in Battery % drained

### 4.3 Prediction

Figure 4 combines the component coefficients with the application statistics to determine the energy cost of selected applications. The energy cost estimated by the profiler came out to 12.9% while the total battery dropped from 76% to 64%. It should be noted that the battery percentage given by the Android OS is precise only to 1% meaning the actual range could be anywhere from 12.9% to 11.1%. Given the general lack of precision in Android battery output and the broad assumptions SApp makes in regards to calculating component coefficients, we are fairly satisfied with the power model.

A follow-on test calculated that 15 additional minutes of Angry Birds would cost 2.57% battery. After executing 15 more minutes of Angry Birds, the power model output 2.84% energy drain due to Angry Birds. Tests of a similar game, Candy Crush, resulted in a prediction that fell within 14% of the actual energy. Tests of the browser fared even better, often falling within 6%. Similar tests for other applications also indicate that basing prediction on previous behavior is a suitable assumption.

## 5 Future Work

The work conducted this this project can be extended in several directions. First and foremost is to increase the accuracy and inclusivity of the components measured. The display test should be opened up to include OpenGL calls into the GPU, measure user interactions with the touch screen, and measure brightness. The WIFI model too should be expanded to include signal strength, bytes transmitted, packets sent and received, and possibly include 3G network calls. Many papers have already been published on WIFI power analysis that future developers could utilize for further refinement [3][2][1][15]. Based off of SApp's results and those of related works, these two areas along with the CPU encompass the largest energy drain and would benefit the most from more refined testing. Other popular device components such as audio, GPS, and Bluetooth should also be incorporated into the tests.

Second, SApp should be made more inclusive to more users by reducing the reliance on

| Name | Time Active (s) | Display Cost (%) | CPU Cost (%) | IO Cost (%) | WIFI Cost (%) | Total cost (%) | % Cost per Minute |
|---|---|---|---|---|---|---|---|
| System Services | 3786.05 | 0.000 | 0.126 | 0.001 | 0.142 | 0.268 | 0.0034 |
| Angry Birds Foreground | 2168.274 | 4.598 | 3.712 | 0.039 | 0.058 | 8.407 | 0.2227 |
| Angry Birds Background | 1617.776 | 0.000 | 0.465 | 0.001 | 0.000 | 0.465 | 0.0175 |
| Browser Background | 2199.242 | 0.000 | 0.041 | 0.056 | 0.032 | 0.129 | 0.0032 |
| Browser Foreground | 1586.808 | 3.365 | 0.128 | 0.085 | 0.014 | 3.592 | 0.1240 |
| Photo Gallery Foreground | 217.875 | 0.420499 | 0.10098 | 2.024935 | 0.006074 | 2.552487 | 0.702922 |
| Candy Crush Foreground | 621.012 | 1.195457 | 0.205183 | 0.219236 | 0.002851 | 1.622727 | 0.156782 |
| Notepad App Foreground | 434.807 | 0.839178 | 0.100313 | 0.005009 | 0.010122 | 0.954621 | 0.13173 |
| Camera Foreground | 154.878 | 0.298915 | 0.006819 | 0.309427 | 0.00338 | 0.618541 | 0.239624 |
| Sapp | 3786.05 | 0.000 | 0.058 | 0.078 | 0.000 | 0.136 | 0.0022 |
| Sum (%) | | 10.717049 | 4.943295 | 2.818607 | 0.268427 | 18.745376 | |

Figure 4: Sample Cost of Various Applications with all Output in Battery % drained

a custom kernel and require rooted devices. The possibility exists for a dedicated developer to exploit some Android API's, alter the polling techniques, or uncover more assumptions in existing, unrooted devices. A power model built in this environment would surely prove desirable by the Google Play market users as power management is currently a hot commodity.

A different direction this project could take is in the manner it interacts with users. SApp currently features little in the way of UI and could be expanded to feature data visualizations and comparisons between several applications. In addition, SApp could use the authority granted by root access to allow users to place limits on resource usage by applications. Rules could be enforced to prevent background applications from wasting CPU cycles if they have not been made active with time thresholds. This should speed up other applications by eliminating wasted CPU cycles.

# 6 Related Work

The mobile device power management research area is enormous, but we will touch on a few of the more recent studies and applications. PowerTutor is very similar to SApp in that coefficients are calculated for a variety of components to create a power model for the device[17]. However, PowerTutor's coefficients are calculated with a power meter in a laboratory environment limiting the number of devices that are supported by the application. WattsOn, created by the Windows 8 team at Microsoft, is another successful power profiling application for use by application developers[10]. As with PowerTutor,

WattsOn was developed in a laboratory environment by measuring specific components with power meters. While very successful in creating power models that do not require alterations to a user's device, maintaining a model for every possible device is unfeasible. In a talk given in Madison on 29 April 2014, Dr. Chandra, from WattsOn, offered that future versions would require hardware vendors to supply component costs as even a company as large as Microsoft could not sustain excessive device testing. The software-only design of SApp allows for a larger number of devices to be incorporated and a custom power model generated for a device.

AppScope takes a similar, modified-kernel approach by monitoring for component hardware request events and charges them to the appropriate application[16]. However, most trials in their publication are limited to less than four minutes where it would be interesting to observe the accuracy in longer tests. Additionally, SApp extends the profiling into forecasting future energy drain.

Carat[12] takes the collaborative approach to power modeling compare the battery drain of an application on one device to that of a community. Carat uses machine-learning to infer what applications are draining battery and offers actions to take such as killing certain application to extend battery life. It is an innovative approach to extending battery life, but does not build a power model to provide detailed information to developers. Also, the community design requires offloading lots of data which may represent privacy issues to some users.

Eprof[13] uses fine-grained approaches to identify energy bugs in source code and even specific lines of code. While interesting, Eprof does not appear to have been updated since 2012 and is not available in the market.

# 7 Conclusion

SApp demonstrates the potential to a software only approach to creating a mobile device power model. This approach provides more inclusion to the number of devices, particularly in the fractured Android market with ten different device manufacturers. SApp could be improved by either reducing the necessity of rooting the phone and installing a custom kernel or by adding to the depth of the component testing.

With three simple steps, SApp produces a power model specific to the device at a certain point in time, gathers data regarding the applications that run on the device, and produces fairly accurate battery costs and predictions. It manages to do this with relatively low overhead and user interaction. SApp assists users and developers by extending battery life through more informed decisions and energy optimized programming.

# References


[1] Rahul Balani. Energy consumption analysis for bluetooth, wifi and cellular networks. *Online]http://nesl.ee.ucla.edu/fw/documents/reports/2007/PowerAnalysis.pdf*, 2007.

[2] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. 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*, pages 280–293. ACM, 2009.

[3] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 21–21, 2010.

[4] Aaron Gingrich. Are you content with your phone's stock battery life?, November 2012.

[5] Google. Android, May 2014.

[6] IBM. Understanding the cpufreq subsystem, May 2014.

[7] Dr. Achim Leubner. Google play: Usage timelines, May 2014.

[8] King Kong Lock. Google play: Powersaver, May 2014.

[9] IMOBLIFE Co. Ltd. Google play: Battery booster, May 2014.

[10] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 317–328. ACM, 2012.

[11] Rahul Murmuria, Jeffrey Medsger, Angelos Stavrou, and Jeffrey M Voas. Mobile application and device power usage measurements. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 147–156. IEEE, 2012.

[12] Adam J Oliner, Anand Iyer, Eemil Lagerspetz, Sasu Tarkoma, and Ion Stoica. Collaborative energy debugging for mobile devices. *Proc. of USENIX HotDep*, 2012.

[13] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. 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*, pages 29–42. ACM, 2012.

[14] Google Play. Google play: Powertutor, May 2014.

[15] Yu Xiao, Petri Savolainen, Arto Karppanen, Matti Siekkinen, and Antti Ylä-Jääski. Practical power modeling of data transmission over 802.11 g for wireless applications. In *Proceedings of the 1st International Conference on Energy-efficient Computing and Networking*, pages 75–84. ACM, 2010.

[16] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *USENIX ATC*, 2012.

[17] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM, 2010.