NEAT: A Novel Energy Analysis Toolkit for Free-Roaming Smartphones

Niels Brouwers  Marco Zuniga  Koen Langendoen
Delft University of Technology
The Netherlands
{n.brouwers,m.zuniga,k.g.langendoen}@tudelft.nl

Abstract
Analyzing the power consumption of smartphones is difficult because of the complex interplay between soft- and hardware. Currently, researchers rely on mainly two options: external measurement tools, which are precise but constrain the mobility of the device and require the annotation of power traces; or modelling methods, which allow mobility and consider explicitly the state of events, but have less accuracy and lower sampling rates than external tools.

We address the challenges of mobile power analysis with a novel power metering toolkit, called NEAT, which comprises a coin-sized power measurement board that fits inside a typical smartphone, and analysis software that automatically fuses the event logs taken from the phone with the obtained power trace. The combination of high-fidelity power measurements and detailed information about the state of the phone’s hardware and software components allows for fine-grained analysis of complex and short-lived energy patterns.

We equipped smartphones with NEAT and conducted various experiments to highlight (i) its accuracy with respect to model-based approaches, showing errors upwards of 20%; (ii) its ability to gather accurate and well annotated user-data “in the wild”, which would be hard to do with current external meters; and (iii) the importance of having fine-granular and expressive traces by resolving kernel energy bugs.

Categories and Subject Descriptors
C.4 [Performance of Systems]: Measurement techniques; D.2.5 [Software Engineering]: Testing and Debugging—Debugging aids

General Terms
Measurement, performance

Keywords
Power monitor, accuracy, mobility, trace visualization

1 Introduction
Short battery lifetimes usually top the list of complaints among smartphone users. This concern has been a driving force behind a series of pioneering studies dissecting the energy consumption of smartphones to identify energy bugs and hidden energy costs. For example, wakelocks have been found to be the source of many energy bugs such as those found in the Facebook- and Android Email apps [17]. With the eprof tool, Pathak et. al. found that in various popular free apps, 65% to 75% of the energy is spent on third party advertisement [16]. There is thus a clear need to analyze the energy consumption of smartphones, but obtaining the fine-grained energy profile required for this type of analysis is not a simple matter.

The tight interaction among a smartphone, the usage pattern of its owner and the specific characteristics of the surrounding environment, determines the energy profile of the device. Each one of these three elements has a rather complex and random behavior that is difficult to capture: smartphones have many hardware and software components interacting constantly; the location, mood and personality of the owner plays a central role in defining usage patterns; and external conditions, such as poor 3G coverage and bright sunlight, affect the operation and the lifetime of the device [19].

Given the central role that power consumption plays in the operation of smartphones, and the complexity of the problem, the research community has shown a strong interest in power monitoring tools. Broadly speaking, these monitoring tools can be divided into two groups: hardware-based and model-based methods.

Hardware-based methods rely on purpose-built tools such as the Monsoon power meter [15]. These types of tools provide a high accuracy and granularity, and hence, they are excellent to obtain a precise power trace for the complex interplay between the hardware and software of a smartphone. The main drawbacks of this approach are first that the phone needs to be tethered to a desk, which hinders the ability to capture the other two important dimensions of energy profiling: usage patterns and varying surrounding conditions; and second that it does not annotate the power trace automatically with the events causing the rising and falling flanks in the power trace. This lack of contextual information makes it difficult to understand and analyze the collected data.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SenSys’14, November 3–5, 2014, Memphis, TN, USA.
Copyright 2014 ACM 978-1-4503-3143-2/14/11 ...$15.00
http://dx.doi.org/10.1145/2668332.2668337
Model-based methods, on the other hand, rely on the internal battery sensors of phones and overcome the limited granularity of these sensors by building energy models on top of them [7,12,16,21]. The advantage of these methods is that they are unobtrusive to users and can work on any smartphone without modification or external hardware. Energy models however are not easy to develop and implement, and more importantly, they have fundamental limitations that affect their accuracy and their (in)ability to observe hidden components. In effect, this means that under many real-world scenarios, model-based methods can only provide incomplete and/or imprecise information about the energy consumption of a phone. These limitations are known in the community, and they are also highlighted on other state-of-the-art model-based methods [7].

Hence, regarding the ability to obtain accurate power traces, we have reached a point where “the challenges posed by rising hardware complexity and variability, motivates the need for increased direct measurement of power consumption” [13]. But, obtaining an accurate raw power trace is not sufficient. In order to understand, analyze and debug the tightly coupled interaction among phones, their users’ patterns and the environments the users operate in, we require a richly annotated power trace.

Within this context, we introduce NEAT: an energy-analysis toolkit that allows fine-grained monitoring and analysis of energy consumption in a portable manner. The cornerstone of NEAT is the insight that we need to combine high-fidelity measurements, captured in situ, with a detailed event trace containing the state changes of the various hardware and software components on the phone. By fusing the two data streams NEAT allows for interactive analysis and visualization of complex and short-lived energy patterns such as OLED panels and CPU wake-up/suspend cycles, respectively. We believe that, overall, our work contributes to the field of smartphone energy analysis in the following way:

- We provide the rationale for combining external power metering with explicit state information to advance the state-of-the-art in energy analysis for smartphones and other mobile equipment. (Sections 2 + 3)
- We present a coin-sized power metering board that fits inside commercially available back-covers made for double-sized batteries. This form factor allows for capturing traces with the user/phone in the loop. The resolution and sampling rate of the NEAT board are 38.15 µA and 2 KHz, respectively. (Section 5)
- We present a software tool that automatically annotates power traces from an external metering board with event data captured on the phone. This tool enables flexible yet powerful post-processing and analysis. (Section 4)
- We showcase the value of our approach by analyzing kernel energy bugs caused by short-lived power spikes, monitoring components whose power draw is a higher-order function of multiple parameters, and by investigating the sampling requirements of events where users interact with the phone’s screen (e.g. scrolling, swiping) as they go on with their daily activities. (Section 6)

We have released the code of our software tool and the schematics of our metering board to be freely available for the community (see http://code.google.com/p/neat-power-toolkit/). We have also released anonymized versions of the traces we captured in situ with NEAT for reference, and to support additional analysis.

2 Background

Power consumption analysis broadly depends on two sources of information recorded during an experiment: (i) a time series of power measurements, showing the total energy consumption of the phone, and (ii) a time-stamped list of events recorded on the phone, tracing relevant components’ internal state changes over time. Colloquially speaking, the power trace depicts “what” is happening, while the state trace explains “why” it is happening. In the following discussion we will place power analysis techniques along two axes, one related to how the power measurements are collected, and the other related to how the phone’s state is taken into account. Figure 1 shows this classification, with NEAT filling a niche not currently occupied.

**Power Measurement** Power traces may be obtained in one of two ways: by sampling the internal battery monitoring unit (BMU), or using an external measurement device. Internal sensors (cf. top left quadrant in Figure 1) have the advantage of being portable, readily available, and not requiring additional hardware or modifications to the device. Unfortunately, their signal is notoriously inaccurate, coarse, and of low frequency. Many BMUs do not measure current directly, instead, they measure voltage and then apply a battery model to obtain a battery level estimate as a percentage of a full charge. Some BMUs only update their readings when their estimate changes, others provide fresh readings every several hundred milliseconds (e.g. the MAX17047 in the Galaxy S3 updates every 175.8 ms).

Alternatively, external power meters (cf. top right quadrant in Figure 1) measure the current draw of the device directly with high accuracy and resolution, usually by placing a shunt resistor between the battery and the phone. Such a setup can be achieved using a digital scope, or by using the popular Monsoon power meter [15]. The latter also provides power directly to the phone, which has the added benefit of providing a constant voltage, which in turn leads to more accurate power measurements. The downside of exter-
The above discussion only deals with issues related to obtaining the power trace, but making sense of this trace “why did my battery die after only five hours?” is not that easy as smartphones comprise many interacting hardware and software components. When no event trace information is available (implicit case, cf. top row in Figure 1) there is little one can do. One option is to use controlled experiments to force the operation of the phone into a known state (or series of states) and measure the power profile of these specific states. Alternatively, one could try to reverse engineer the information contained in the overall power trace by “spotting” the presence of specific power profiles. None of these methods provide accurate and insightful information about high-level actions though.

As an alternative, systems have been proposed that track the states of the individual hardware components (e.g., display, CPU, network) over time and combine system state with measured power to estimate the consumption of the individual elements. These power modelling approaches overcome the limitations of the low accuracy and update rate of BMUs by estimating power consumption from system state, which typically can be measured at a higher frequency (e.g. 100 Hz). Power models may be learned either offline using an external power meter [22], or on-line using a BMU capable of sensing current [7].

The advantages of power modelling are that it works on any device, is fully mobile, and allows for experiments with the end-user in the loop. Unfortunately there are several limitations as well. First, as pointed out by McCullough et. al. [13], the linear regression methods used on energy models are inaccurate (because power consumption is not linear), and more complex regression methods would not improve accuracy significantly. Second, the increasing complexity of hardware is leading to a growing number of hidden device states that are not exposed to the operating system (which energy models utilize as input). For instance, a number of types of system state are unobservable from the perspective of an Android application, such as the power state of the 3G modem, or the tail state of the GPS chip [17]. Third, the system state cannot be observed during those times when the main CPU is in suspend, and therefore power can not be computed. Not accounting for the energy spent during those times when the CPU is in suspend mode can lead to gross miscalculations. We show in Section 6.3 that for casual users, over half of the energy spent during normal smartphone operation occurs when the CPU is in suspend mode.

3 NEAT

Considering the limitations of the state-of-the-art, we propose a new monitoring tool consisting of hardware and software components, called NEAT. For the hardware part, we designed a mobile, high-speed power metering board that can be attached to a typical smartphone. The board is small enough to fit inside slightly larger battery covers designed for double-sized batteries, which makes it completely unobtrusive to end users, and allows us to capture high-speed (2 KHz) traces from every-day use. Figure 2 shows how our board fits in with the state of the art: compared to the Monsoon power meter we trade off a bit of accuracy for a major step in usability (tethered vs. mobile operation), while we achieve results that are much more accurate than those obtained with BMU-based power modeling at a slight reduction in functionality (the buzzer is disabled for synchronization purposes, see Section 5.1).

To make sense of the hours-long traces obtained in this way, we developed software that automatically annotates power tracess with system state to make in-depth analysis tractable. Events from the Android kernel and user-space logging facilities are recorded concurrently with the power measurements, and later fused into the power trace. Figure 3 illustrates how this would be applied to a specific use case. Figure 3(a) shows a raw power trace as would be obtained from the measurement board. In this example, the phone is performing an HTTP request to an online service over 3G internet. Several hard- and software components inside the phone are working together to create the observed pattern, but without context, this trace is very difficult to interpret, severely limiting its use. For comparison, Figure 3(b) was generated using NEAT. Here, the vertical lines represent events collected from the phone at the same time the power trace was recorded. Using techniques described in Section 4, NEAT is able to annotate the trace and show when the CPU was awake, how long the 3G ramp-up took, and show energy tail of the modem. Moreover, because NEAT is fully scriptable, it is possible to compute statistics such as duration and energy for each of these operations, and do it over hours-long traces in an automated way.

3.1 Installation and Use of NEAT

NEAT has three main components, shown in Figure 4. The Power Meter is a high-speed (2 KHz) power measure-

![Figure 2. Advantages of NEAT relative to the state-of-the-art. The NEAT mobile power meter sacrifices some accuracy compared to a Monsoon, but is portable (1). Our meter is more accurate compared to power modeling, but it requires soldering to the device (2).](image-url)
event board that records its data onto a micro-SD card. It is powered by a separate battery, and is installed inside the target smartphone. For synchronization purposes, it has a trigger input that needs to be wired to a hardware output of the phone. In our experiments we desoldered the phone’s vibrator, drilled a small hole in the case, and wired this connection to the NEAT board. More detailed instructions about the hardware installation are provided in Section 5. The schematics of the board are freely available (see http://code.google.com/p/neat-power-toolkit/). The Event Logger records events from the phone. If recording kernel events is needed, this application requires root access. Finally, the information coming from the power meter and the event logger are input into our Trace Viewer desktop application. The code related to the Android App and the trace viewer, including the code used in our evaluation, is also freely available.

From the perspective of the end-user, the operation of NEAT is straightforward. The batteries of the phone and the meter should be charged via their respective Micro-USB connectors. Then, to log information, the user pushes the button on the meter board (Figure 4, top left) to start recording power data, and start the Event Logger (Figure 4, bottom left). At the end of a recording session, both can be stopped again, and their data can be collected from the micro-SD card and the phone’s internal storage respectively.

4 Power Analysis

A typical analysis of complex power traces requires three primary functions: a correct mapping of kernel and user space events onto the power trace; visual inspection of the data to zoom into suspicious or abnormal behavior; and data processing to extract statistics and quantify how serious and pervasive the abnormal behavior is. In this section, we describe how our power analysis tool implements these functionalities. At this point we assume that we already have an accurate power trace. The tool is able to read power traces files produced either by a Monsoon power monitor, or by our custom built mobile power metering board (see Section 5).

Figure 5 shows a screen shot of the viewer application. Our tool provides two views of the data. First, a time-series plot showing the power, current, voltage and trigger channels, with the events overlaid on top of them (only the power is shown on the left part of Figure 5). Second, an event trace view showing a list of timestamped events (right). The two views are linked, so that browsing the time line in either window updates the other automatically.

As for the data processing, instead of exposing the required functionality into a potentially complex UI, we opted to take the simple route of making the tool fully scriptable by exposing a programming API to the Rhino JavaScript interpreter [18]. Users write small scripts to automate different steps of the analysis process, which can be run from a pull-down menu in the UI. Alternatively, our tool can run headless, which is more convenient for batch processing.

4.1 Trace Mapping

As we will describe in the next section, the Data Acquisition process needs to provide two traces to our Power Analysis tool: a power trace, and an event trace containing kernel-
and user space events. Our first goal is to map the kernel and user events onto the power trace. There are three different clocks that need to be synchronized: (i) Kernel events are timestamped with kernel uptime in milliseconds, which is a continuous clock that does not increase while the phone’s main CPU is in suspend mode, (ii) LogCat events are timestamped with the real-time clock (RTC) time, which increases during suspend mode, and (iii) the power trace is timestamped by the internal clock of the power meter, which could be the Moonson meter or our own embedded meter\(^1\).

Our trace mapping method is depicted in Figure 6 and consists of two steps: first the two event streams (kernel and logcat) are mapped onto a single time-line, and then this single time-line is mapped onto the power trace. Although the principle is generalizable across phones, different setups may require slightly different parameterizations of the individual steps. For this reason, we have implemented the bulk of the algorithms in Java, but drive them from scripts optimized for specific devices.

**Synchronizing Kernel- and User Space.** During normal operation, the CPU of a phone will go into suspend mode when not in use, at which time the clocks of the two event streams diverge. We map kernel events onto the global time line by splitting the kernel event trace into segments and repositioning each segment to the appropriate location inside the user-space stream. Figure 6 illustrates the process. The splitting points are determined by one or more kernel messages indicating that the CPU is coming out of-, or going into suspend.

Each segment starts with a number of messages from the Android kernel as it is coming out of suspend. Although many of these messages are fairly standard across devices, differences may still exist. For example, some platforms will report the cause of the wake-up (e.g., button, clock, modem), while others do not. One particular message that always appears after coming out of suspend records the current user-space (RTC) time. This RTC time can be used to compute how far in time each segment needs to be shifted to line it up with the LogCat time-line.

**Synchronizing Event- and Power Traces.** Time synchronization between event- and power traces depends on so-called anchor points, which are timestamped events that are observable in both traces. In Section 5.3, we describe in detail how we use the phone’s vibrator to obtain such anchor points. For now it is sufficient to have the following basic description: the phone’s vibrator is executed periodically, and these events can be logged in the event trace and they can also be observed in the power trace (blue traces in Figure 6). Synchronization can then be achieved by matching anchor point events to the corresponding rising flanks in the power trace, and performing a linear regression on the events between two consecutive anchor points to map them onto the power trace. In our case, matching anchor point events to the corresponding rising flanks is made challenging by the fact that the phone’s vibrator may also be triggered by sources other than the logger, e.g., incoming calls, email, and so on, so that rising flanks in the power trace typically outnumber the anchor points observed in the event trace.

We solve this mapping problem using a greedy approach. Consider a sequence of logged synchronization events \(E = \{e_1, \ldots, e_n\}\), that needs to be mapped onto a sequence of rising flanks \(F = \{f_1, \ldots, f_m\}\), with \(n \leq m\). We need to construct a mapping \(\mathcal{M} : E \rightarrow F\) that matches each event with its corresponding flank. Assuming that a trace is correctly matched up to \(e_i\), our greedy iteration step selects a flank \(\mathcal{M}(e_{i+1}) \in F\) that minimizes

\[
(t(e_{i+1}) - t(e_i)) - (t(\mathcal{M}(e_{i+1})) - t(\mathcal{M}(e_i)))
\]

Figure 5. The NEAT trace view application.

Figure 6. Aligning kernel- and logcat event time lines. Kernel events are timestamped with kernel uptime, a clock that does not increase when the CPU is suspended. The first step of the synchronization process is to split the kernel trace into sections that correspond to CPU wake periods and map these onto the global time line.

\(^1\)The choice of time stamps for kernel- and user-space events is a property of the Android OS, and sadly out of our control.
where $t(.)$ denotes the timestamp of a given event or flank. The concept is illustrated in Figure 7. Because $t(e_i)$ and $t(M(e_i))$ are monotonically increasing, in practice only very few subsequent flanks $M(.)$ need to be considered.

### 4.2 Visualization

Making sense of experimental data often starts with visual inspection. Visualization tools are even more important when the user is in the loop because the experimental data is affected by the users’ patterns and environments. The unique behavior of users influences the way different software components work, including system services, device drivers, and applications [8]. Our tool has several facilities for visually annotating labeled traces, in order to highlight certain event types, regions of interest, and so on. Our tool supports exporting the annotated graph to PDF for inclusion in publications; all graphs in this paper were generated this way.

**Tags.** Tags are matching functions for events, and can be used to highlight or filter out specific events. For example, one may want to look at events from a certain application only, or zoom in on a certain type of error message related to a known bug. A user can tag events of interest, and then choose to hide all untagged events in the interface, making it easier to browse the data without being distracted by unrelated ‘noise’. Figure 5 shows how tags are shown in the event list in the UI (right). Tags are displayed next to the event, and may optionally have a name and color for this purpose.

**Regions.** In the same way tags match events, region tags are matching functions for sections of a trace. Region tags consist minimally of a start- and end tag, and they may be used to annotate the trace and help with visual inspection. For example, in Figure 5 several regions are shown that indicate the CPU being awake, the time when the Wi-Fi chip is busy with an access point scan, and the head- and tail times surrounding the scan.

Another example is marking which apps were active at which time. This is illustrated by Figure 8, which shows a trace from a user switching between different screens of the game *Wordfeud*. Here, regions are added one-by-one by a piece of script that matches application resume-and-pause events. The figure shows that the chat screen (blue region on the right of Figure 8) consumes more energy compared to the game board, presumably because of user interaction (scrolling, typing versus looking for a good move to make), and because this screen uses lighter colors that consume more energy.

**Annotations.** Finally, annotations help clarify particular points of interest in the trace. For example, Figure 9 shows a part of a user trace where the CPU is woken up by the Wi-Fi chip to process incoming data. Annotations are positioned by timestamp and value, where the value may be power, current, or voltage.

To illustrate how this may be achieved programmatically, Listing 1 shows a script that adds the ‘Wi-Fi Wake’ arrows shown in Figure 9 to a trace. Line 2 gets a list of kernel events that indicate a CPU wake-up caused by the Wi-Fi chip (*bcmsdh_sdmcc* is a part of the Broadcom Wi-Fi driver). Then, for each such event, an ArrowAnnotation object is created with a start- and end coordinate, and a text label. To make the arrow point down and to the right, the start coordinate is positioned 150 ms to the left, and 500 mW upwards.

### 4.3 Processing

The same scripting principles used in visualization can also be applied to automatically process data sets from collected traces. As an example, let us characterize the energy consumption of the OLED panel, a notoriously difficult device to model. We built a full-screen Android application that fills the screen with a single color, cycling through different colors in the RGB color space every 400 ms, and out-

---

**Figure 7.** Greedy mapping process. Given a partial mapping up until element $i$, $M(e_{i+1})$ is chosen such that the difference between $t(e_{i+1}) - t(e_i)$ and $t(M(e_{i+1})) - t(M(e_i))$ is minimized.

**Figure 8.** Using regions to show which app was visible at what time in the trace. Real-world trace from a user playing the game *Wordfeud*.

**Figure 9.** Annotations, such as arrows, may be used to draw extra attention to certain types of events. In this case, the CPU waking up in response to the Wi-Fi chip.
Listing 1. Adding arrows to a power trace.

```java
// Events that indicate wake-up caused by Wi-Fi
var e = events.filter('Resume caused by IRQ 162, bcmsdh_sdmmc');

// For each event, create an annotation
for (var i=0; i<e.size(); i++)
{
    // Get event timestamp in milliseconds
    var time = e.get(i).getTime();

    // Create annotation
    model.addAnnotation(
        new ArrowAnnotation(
            'power', // Select the power scale
            time - 150, 1.2, // Start coordinate
            time, 0.7, // End coordinate
            'Wi-Fi Wake' // Label
        )
    );
}
```

Listing 2. Post-processing script for the OLED experiment described in Section 6.1.

```java
// Obtain a list of color change events
var e = events.filter('COLOR CHANGE');

// For each color, compute average power
for (var i=0; i<e.size(); i++)
{
    // Event timestamp
    var time = e.get(i).getTime();

    // Compute average power over 200ms
    var avgPower =
        power.getPower().getAverage(
            time + 100,
            time + 300
        );

    // Print text and power
    print(e.get(i).getText(), avgPower);
}
```

Putting a log message every time it does so. Figure 10(a) shows a power trace collected from a Galaxy S2 smartphone during a run of this experiment. The aim is to obtain the average power consumption of each (color) screen.

With experimental traces collected and correctly synchronized, the script shown in Listing 2 can be used to extract the color values and corresponding power measurements. The script extracts the necessary information in the following way: (i) the `filter` method is applied to obtain a list of all color change events (lines 2-3), (ii) for each RGB event, the average power is computed between 100 ms and 300 ms after the color change event (lines 11-15). In the script, `power` is a reference to the power trace, and `getPower` returns the power series of that trace.

A major advantage of this approach is that even though this is a controlled experiment, there will still be influences such as background applications that cannot be completely eliminated. By cycling through our parameters space several times and collecting multiple samples for each parameter assignment (each color, in this example) we are able to remove outliers caused by such interference and increase both the quality and confidence of our data. Figure 10(b) shows the resulting data for a Galaxy S2 smartphone. In Section 6.1, these results are used to show the comparative advantage of NEAT with respect to model-based methods.

5 Data Acquisition

One of our main goals is to design a tool that overcomes the fundamental limitations of model-based approaches: lack of accuracy and inability to expose “hidden” events. To achieve this goal, we rely on an accurate and fine grained power trace. Thus far, we have assumed that our Power Analysis software have readily available such a power trace plus the event trace and anchor points to synchronize both traces. In this section we describe how we obtain the required traces and anchor points.

5.1 The NEAT Mobile Power Metering Board

A major drawback of external power meters like digital scopes or the Monsoon power monitor [15] is that they constrain the mobility of the device. But the level of accuracy and granularity achieved by these meters is necessary for NEAT to work on-board. Inspired by the Monsoon meter, we designed a custom power metering board that fits inside commercially available back-covers designed for double-capacity batteries. These type of covers are available for most popular smartphone models. This setup makes the phone slightly bulkier, but it has the key advantage of resulting in a fully usable smartphone that can be carried by end-users.

In this subsection we will first introduce the components of the board, then we will explain the tradeoffs and design decisions made to achieve a sufficiently precise yet compact meter, and finally we describe the usability and the steps required to connect our board to a smartphone.

Measurement Board. Power is determined by the voltage and current flowing through a device. To capture these signals, we used an STM32F373CB, a 32-bit ARM Cortex M4 microcontroller with 32 KB of RAM. Two 16-bit analog-to-digital converter (ADC) channels are used to measure battery voltage and current draw. A third auxiliary channel, called trigger, is used to synchronize the power and event traces. The meter has its own Li-Po battery, between 450-600 mAh depending on the available space, and it can provide 10.5-14 hours of continuous measurement\(^2\). By fitting an extra battery, instead of powering the monitor from the phone battery, we avoid adding measurement noise. Data is stored on a Micro-SD card, which provides ample space for multi-day experiments.

To increase the robustness of the installation, we used a 3D-printed plastic cover to fill out the extra space around

\(^2\)The power consumption of the meter itself is 42.8 mA on average, as measured using a Monsoon Power Monitor
the metering board and battery. The size of the board (39x16x5 mm) and the size of the battery are small enough to fit into the smallest phone in our collection, a Galaxy S smartphone. The assembled device is shown in Figure 11.

Range and resolution. To provide an adequate resolution, the Monsoon meter has two shunt resistors, “coarse” and “fine”, which it switches between automatically (depending on the magnitude of the current). The “fine” shunt is used to capture currents $\leq 40$ mA. In the interest of simplicity and to save space on the board, we aimed for a design that has a single shunt. The first step was to identify a sensible range (the Monsoon’s range is $\pm 4.5$ A). We do not consider a large fraction of the negative range, since a negative flow indicates that the phone is being charged and this operation is not of interest for our purposes. For positive current flows, we consider an upper value of 2 A, which we found sufficient for typical smartphone platforms. Values outside this range are clipped. We found this upper limit empirically using a Monsoon power monitor, and validated it using the data set from Section 6.3, which showed that less than 0.1% of the data was above 2 A. With a single 75 m$\Omega$ shunt we provide a resolution of 38.15 $\mu$A. Table 1 compares the technical specifications of the NEAT board with the Monsoon Power Monitor [15], which is the de-facto standard for smartphone power analysis at the time of writing, and some first-generation portable meters discussed in Section 7.

It is important to remark that NEAT’s hardware design can be improved by borrowing some of the ideas presented in the Nemo power meter [23], which is aimed at sensor networks. Nemo has a multi-shunt resistor circuit that allows a bigger dynamic range (i.e. no need to clip high current values as in NEAT), and utilizes more advanced components to provide a sampling rate above 8 KHz. For the current needs of smartphones however, we felt that the single shunt and a 2 KHz sampling rate were reasonable choices to tradeoff accuracy for size and simplicity.

Sampling Rate. As described by Dong and Zhong [7], multitasking systems, such as smartphones, require a sampling rate that is no lower than 100 Hz. In our case, the on-chip ADCs provide a high sampling rate of up to 50 KHz, but the signals need to be down sampled before being written out to the SD card. The down sampling is required because SD cards exhibit long write latencies when crossing sector boundaries as the new sector is erased. Hence, every time a sector is crossed, the controller has to buffer samples during this time, which makes memory the limiting factor on the sampling rate. In our design the sampling rate was reduced to 2 KHz; which is lower than the 5 KHz provided by the Monsoon power meter, but significantly higher than the 100 Hz sampling rate provided by the state-of-the-art model-based methods. Lowering the sampling rate to 2 KHz reduces data loss to a minimum, but it cannot be ruled out entirely. For this purpose, samples are grouped into 128-byte blocks of 21 samples, and tagged with a 16-bit sequence number which can be used to detect missing blocks. In practice, we found that the amount of missing data was negligible (see Table 3). At a rate of 6 KB/s, a single 4 GB card can store about 98 hours of measurement data.

Usability. To install a NEAT board one needs to solder the connectors shown in Figure 12. The ‘battery+’ and ‘ground’ pads are connected to the phone’s battery, while the

---

3While we did not investigate the source of these high currents, we hypothesize that they occur when more than one component exhibit extreme power spikes, for example when the screen is on showing bright colors, at a time when the modem is performing a hand-over between towers.
Table 1. Properties of the NEAT power meter compared to related devices.

<table>
<thead>
<tr>
<th>Feature</th>
<th>NEAT</th>
<th>Monsoon</th>
<th>BattOr</th>
<th>Holleis et. al.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>-0.5 A - 2 A</td>
<td>±4.5 A (±40 mA fine)</td>
<td>0-1 A</td>
<td></td>
</tr>
<tr>
<td>Resolution</td>
<td>38.15 µA</td>
<td>286 µA (2.86 µA fine)</td>
<td>16-bit</td>
<td>16-bit</td>
</tr>
<tr>
<td>ADC</td>
<td>16-bit</td>
<td>16-bit</td>
<td>10-bit</td>
<td>10-bit</td>
</tr>
<tr>
<td>Sampling Rate</td>
<td>2 KHz</td>
<td>5 KHz</td>
<td>‘hundreds’</td>
<td>500 Hz</td>
</tr>
<tr>
<td>Size</td>
<td>39x16x3 mm</td>
<td>194x152 mm</td>
<td>50.8x38.1 mm</td>
<td>68.7x53.3 mm</td>
</tr>
</tbody>
</table>

‘load’ pad corresponds to the phone’s positive terminal. To facilitate the soldering to the phones’ battery pads, strips of copper tape are inserted between the battery and the phone’s connector, isolated with Kapton tape. The ‘trigger’ pad is used for hardware synchronization, and its connection is explained later in this section. The pads for connecting the board’s own battery are on the bottom left. The USB port is used to charge the Li-Po battery, perform calibration, and control the on-board RTC clock settings. Once installed, the modified phone can be handed out to end-users with the instruction to turn on the power meter by pressing the button on the board, and turn it off by pressing the button again. A single two-color LED provides feedback about whether the device is currently measuring.

5.2 Event Logger

To collect event traces from the phone, we developed an Android application that runs alongside the power meter and records events from both the Android kernel and user-space logging facilities. Figure 13 shows a screenshot of the app.

Kernel Events. The Linux kernel is an important part of the Android stack, and as such has a big influence on power consumption. The Kernel log contains information about the CPU wake-suspend cycle, debug information from drivers, and so on. As we will illustrate in Section 6.2, inspecting the kernel log is crucial in tracking down certain subtle energy bugs. The event logger records kernel events by reading from /proc/kmsg. Since this operation requires root privilege, the device must be rooted and the logger must be granted root access for this feature to work. Each event is timestamped by the device’s uptime clock in microseconds.

User-Space Events. A second valuable source of event data is LogCat, Android’s user-space logging facility. All applications running on an Android smartphone may use LogCat for debugging, by generating events using the Log class. Each event consists of a tag indicating the application or task, a level (e.g., debug, information, or verbose), a string of text describing the event, and the PID of the process that generated it. The event logger automatically queries process information for PIDs it has not encountered before and inserts that into the log. This enables the trace viewer to match events to the applications that generated them.

5.3 Synchronization

Methods that leverage the internal battery-sensor of phones do not need to synchronize the timestamps of power and event traces (because they use the same clock). However, if an external meter is used, the power and event traces use different clocks. Small embedded microcontrollers such as the one used for our monitoring board tend to suffer from clock drift. Hence, as time passes by, the power and event traces get out of sync.

Hardware Trigger. To overcome this problem, we attach a hardware output on the phone to the “trigger” channel on the power meter. Both the Monsoon, as well as our mobile power meter support such a channel. In most cases we use the phone’s vibrator output as the synchronization signal. In this way, triggering the vibrator in software will provide an entry in the event log, as well as a rising flank in the power trace. These two events serve as anchor points for synchronization. Please note that while this method sacrifices the vibration capability of the phone, other device-specific alternatives exist. For example, the Galaxy Nexus has a free GPIO pin on the dock connector that is relatively easy to solder to. Driving this pin from user-space however requires disabling dock functionality and implementing a device driver. Given that the vibrator method is device independent and relatively straightforward, we considered it a more reasonable choice. Connecting the vibrator’s output to our board is a relatively simple procedure. It requires desoldering the vibrator’s leads and soldering them back to the NEAT board (and possibly drilling into the case to bring out the wires).
Table 2. Four Samsung devices used in the display energy modelling experiment.

<table>
<thead>
<tr>
<th>Device</th>
<th>Screen Type</th>
<th>Size</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaxy S</td>
<td>Super AMOLED (Pentile)</td>
<td>4&quot;</td>
<td>480x800</td>
</tr>
<tr>
<td>Galaxy SII</td>
<td>Super AMOLED Plus</td>
<td>4.3&quot;</td>
<td>480x800</td>
</tr>
<tr>
<td>Galaxy Nexus</td>
<td>Super AMOLED Plus</td>
<td>4.65&quot;</td>
<td>1280x720</td>
</tr>
<tr>
<td>Galaxy SIII</td>
<td>Super AMOLED Plus</td>
<td>4.8&quot;</td>
<td>1280x720</td>
</tr>
</tbody>
</table>

Generation of anchor points. The anchor points required to map the events onto the power trace are generated by the event logger app. This app enables periodically the vibrator for a short period of time (<100 ms) and logs this event. Since turning on the vibrator causes a rising flank in the power trace’s trigger channel, this event can be observed in both traces. These synchronization events are spaced with a random distribution between 10-20 seconds. This randomization helps the fitting process described in Section 4.1.

6 Evaluation

In this section we present three use cases to highlight the benefits of our approach. First, we demonstrate the importance of having an accurate meter by profiling the energy consumption of OLED panels and showing that model-based methods lead to large measurement errors. This use-case exemplifies claim (2) in Figure 2 (improvements over model-based methods). Second, we showcase the advantages of having a precise portable meter, together with the friendly software analysis tool, by identifying two kernel power bugs related to WiFi scanning. The portability of our tool and the explicit tracking of the phones’ states support claim (1) (improvements over tethered meters), while the on-board access to fine-grained traces provides further support to claim (2). Third, we use NEAT to monitor the energy patterns of various users while they go on with their normal daily activities. We believe that this accurate and personalized level of energy monitoring will help the community to further improve the energy efficiency of smartphones.

6.1 Display

One of the key advantages of NEAT is that it overcomes the accuracy limitations of model-based methods. We highlight this advantage by comparing the actual energy consumptions of OLED displays with the most advanced models available in the literature. As we will see, the measurement errors are not negligible and can reach values above 25%.

State of the art models for OLEDs. Displays are one of the major sources of energy consumption on today’s smartphones, which makes them a primary target for power profiling. Samsung uses AMOLED panels in their flagship ‘galaxy’ devices, whose power draw is difficult to model due to gamma correction and dynamic color tuning [6,14]. These effects eliminate the additive and linear properties found on other displays. The difficulties of modeling OLED displays are known and mentioned in other studies [12,17].

To the best of our knowledge, the most recent and accurate model has been proposed by Mittal et. al. [14]. This model uses a 16x16x16 look-up table that represents the colors of a discretized RGB cube. Each entry contains the measured power consumption of filling the entire screen with the corresponding RGB color. To estimate the energy cost of displaying a given image, the method calculates first per-pixel costs and then sums these values over the image. In [14], the power consumption for each element of the cube is collected using a Monsoon power meter. Recreating this model requires a large amount of data collection and processing (4096 measurements are required). With NEAT this is a relatively simple matter. We use the method and scripts presented in Section 4.3 to generate the 16x16x16 RGB cube.

We evaluated the accuracy of this model using a data set of 99 images. These images include 19 images from the same dataset used in [14] (not all images were available to us), the Kodak Lossless True Color Image Suite [3], 39 splash screens from popular Android applications, and 20 application screen shots. The data set includes dark, bright, and colorful images, as well as several full-color photos. We evaluated the four devices listed in Table 2. The Galaxy SII and SIII employ a system called content adaptive brightness control (CABC) that dynamically changes the brightness of an image depending on its contents. We evaluated our model both with- and without CABC enabled. The ground truth was obtained by having our app cycle through these images and measuring the device’s power draw.

The NEAT advantage. Figure 14 shows the cumulative error distribution for each of the devices. We can observe that, depending on the device, the maximum error ranges from 5% to over 25%. We also found that CABC greatly degrades the accuracy of the estimates, suggesting that the model should be updated to include this feature. Our findings match those found in [14], where the authors tested the Samsung Focus and found a comparable error range, between 0% and 15%, for the 30 images in their set. Overall, this use case conveys two important points. First, the flexibility of our scripting tool to perform evaluations that may otherwise be complex (Listing 2 allowed us a quick evaluation of four platforms with different configurations). Second –and more important–, even the most advanced energy-models struggle in keeping up with the increasing hardware complexity of smartphones.

6.2 Dissecting Power Consumption of Background Wi-Fi Scanning

In the second use case, we show how NEAT’s high sampling rate, explicit consideration of the phone’s state and visualization are helpful in identifying energy bugs. Due
to the lower sampling rate of energy models some of these bugs would have been unlikely to be detected, and without a portable accurate meter, we would not have been able to perform these experiments with end users in the loop.

Cost of WiFi Scanning. Wi-Fi localization services such as Google’s geolocation service, and Skyhook are a widely popular alternative for energy efficient localization. At the heart of these services lie Wi-Fi scans. In this subsection, we show how we use NEAT to improve the energy efficiency of Wi-Fi scans and to validate our results on end users.

Since the natural state of a smartphone is suspend mode, in which the CPU is turned off, the majority of the scans include a CPU wake/suspend overhead in addition to the scan operation itself. Figure 15(a) shows the three phases of a Wi-Fi background scan: device wake-up (head), the scan itself, and finally the period between the scan completing and the device going back to suspend (tail). Notice that the head and tail overheads account for a large share of the energy consumption of a scan. Each of these phases offers opportunities for energy savings, and will be discussed in turn.

In [4], the authors use a linear model to capture the cost of incremental Wi-Fi scans, which reduce energy costs by scanning only a fraction of the channels, instead of scanning all of them. With NEAT we show that while the cost of incremental scans is indeed linear with the number of scanned channels, fine-grained measurements expose a large variance. Figure 15(b) shows a scatterplot where each point represents a single access point scan. While the total energy consumption (red crosses) shows the expected linear trend in terms of channels scanned, there is also a surprising amount of variance that cannot be readily explained. In fact, plotting the scan operation separately (shown in green) shows that the overhead dominates the overall energy consumption. As is evident from the figure, this removes most of the noise, indicating that the wake-up/suspend overhead is the source of the variance.

Device Wake-Up. The screen and CPU are turned off during suspend, so applications that want to perform some processing task during these periods, such as sampling sensors or checking for new messages, must explicitly request the device to come out of suspend. This is accomplished by setting an alarm that triggers a wake-up when it expires. In Figure 15(a), we can see two suspicious behaviors during the wake-up phase. First, the time between the CPU waking up, and the start of the scan operation is surprisingly long. Second, there are several energy spikes that occur during this time. We were able to identify two bugs in the Android kernel that are the cause of these behaviors.

The first bug (early wakeup) is due to the fact that the CPU wake-up process relies on an interrupt generated by a real-time chip (RTC), which has a resolution of whole seconds. This means that even though alarms may be scheduled with millisecond resolution, the actual wake-up moment is rounded down to the nearest second, resulting in an ‘idle time’ of \( t - \lfloor t \rfloor \) seconds. Since \( t \) is randomized, we should expect this idle time to be uniformly distributed between 0-1 s. However, we found a bug in the alarm driver\(^4\) where during the calculation of the wake-up moment, the time is rounded down twice, so that the idle time is actually between 0-2 s, causing the large variance seen in Figure 15(b).

The second bug (spikes) occurs because when the RTC chip fires its interrupt and the CPU comes out of suspend, the driver fails to acquire a wake lock\(^5\) for the duration of the idle time. This causes the processor to attempt repeatedly to go back into suspend mode. At each attempt, the alarm driver returns an error because it has an alarm to deliver in the near future (PM: Device alarm failed to suspend: error -16), causing the kernel to roll back the suspend operation. This process repeats until the alarm expires, causing the observed power spikes. We have reported both bugs to Google [1,2].

The implementation complexity of Android prevents us from fixing these problems in the kernel, but we can reduce the overhead by detecting the start of the idle period and performing our access scan right away instead of waiting for the wake-up alarm to expire. We implement this by setting a type of repeating alarm that does not cause the CPU to wake up when it fires. Instead, if the alarm expires during CPU suspend, it is delivered to the application at the moment when the CPU comes out of suspend (much like how pending inter-

\(^4\)The alarm driver is part of the Android kernel and is the component that implements the alarm functionality described earlier.

\(^5\)A wake lock is a type of lock that prevents the CPU from going into suspend as long as at least one is held by an application or driver.
To verify that our modification does not interfere with the correct operation of the Wi-Fi driver, we outfitted one of the phones in our test pool (Section 6.3) with this driver and handed it to a test subject that relies on Wi-Fi for internet access. No problems were reported after several weeks of operation, indicating that this modification is safe to make.

We then compared this user’s power traces from before and after the modifications were made. When a phone is connected to the internet, the main CPU has to wake up whenever there is incoming data so it can be handled by the relevant user-space application. This can be observed in the trace as a series of short CPU wake periods that start with a kernel message indicating that the CPU was woken up by the Wi-Fi chip, Figure 16(c) shows a histogram of the power consumption of these short-lived wake periods. Power traces from after the modification show a reduction in median power consumption of 31%.

It is important to highlight three key points in our Wi-Fi debugging process. First, the power spikes observed during device wake-ups are in the order of a few milliseconds in length (most of them are less than 3 ms), which require sampling rates above 333 Hz. Energy-models are unlikely to capture these short-lived events in an accurate manner. Second, our visualization interface greatly improves the understanding of the problem through visual inspection, and allows us to quickly pin-point the issues. Third, the ability to perform precise in situ experiments allows us to provide an accurate assessment of the actual gains experienced by an end-user.

### 6.3 User Study

A major strong feature of NEAT is that we are able to measure power consumption ‘in the field’ from real end-users, at high sampling rate. To highlight the usefulness of this feature, we fitted a number of Galaxy Nexus smartphones with our power meters, and handed them to users who were asked to use them as their primary device for a number of days. Together these seven test subjects collected over 320 hours of data. Table 3 provides an overview of the volume and quality of the data. We found that the loss very little data (0.0039%), and only a small number of samples were clipped because they exceeded the 2 A ceiling of our measuring range (0.0005%). Based on our data, we make a number of observations.

| Table 3. The data set collected during the user study. |
|-------------|-------|-------------|-------------|
| User 1      | Volume [h] | Missing [%] | Clipped [%] |
| User 2      | 34.62   | 0.0057      | 0.0000      |
| User 3      | 9.74    | 0.0000      | 0.0001      |
| User 4      | 52.09   | 0.0004      | 0.0000      |
| User 5      | 64.53   | 0.0135      | 0.0008      |
| User 6      | 27.14   | 0.0003      | 0.0028      |
| User 7      | 33.08   | 0.0018      | 0.0000      |
| Total       | 320.05  | 0.0039      | 0.0005      |

**App Usage.** A question asked by many smartphone users, is which apps on their phones use the most energy. NEAT allows us to partition the power trace based on which apps where active at which time, and thus compute the total energy consumption for each app. In addition to app usage, we also computed how much energy was spent while the CPU was suspended, and the total energy spent on background

### Figure 16. Analysis of the wake-up (top) and suspend (middle) overheads. We also tested our tail optimization in a real-world setting (bottom).
processing. Figure 17 shows this usage for two of our users. User 1 mostly uses the device for communication, social media, and casual browsing. Hence, most of the energy is spent in the suspended state. User 6 is a power-user, and has a much more diverse profile, with only 15.75% of total energy spent in suspend mode.

Sampling Rate. NEAT provides in situ, high-frequency power measurement. To illustrate why a high sampling rate is needed, consider Figure 18(a), which shows a power trace where the user was browsing the internet. Initially, when the user is looking at the web page, power consumption is 'relatively' stable at around 1700 mW, but as soon as the user starts interacting with the device (e.g., scrolling), power draw soars by up to 100% in a highly volatile pattern.

This behavior is caused by the constant updating of the screen contents (e.g., due to user interaction, animation, or movie playback). To illustrate this, Figure 18(b) shows the Fourier transform of the full browsing session, which lasted 68 seconds. In the frequency domain, a strong peak is found at 60 Hz, with its corresponding harmonics at 120 Hz and 180 Hz. The 60 Hz peak corresponds to the refresh rate of the screen. In other words, frequent display refreshes cause a significant increase in power consumption.

From this we can make three conclusions. First, from the perspective of app developers, if energy efficiency is a design consideration, animation should be kept to a minimum. Second, power consumption of foreground apps is strongly influenced by how users interact with it. Third, in order to accurately capture these energy consumption patterns, a minimum sampling rate of 120 Hz is needed (Nyquist Sampling Theorem). And a truthful representation of the entire spectrum would require sampling rates that are upwards of 400 Hz. This required sampling frequency is higher than what is typically offered by built-in current sensors, which provide a maximum of 100 Hz. On the other hand, other mobile meters can provide sampling rates of 500 Hz (Table 1), but they have lower resolutions and ranges, can influence the measured energy signal because they use the same battery as the phone, and they are too big to fit them unobtrusively inside the phone.

6Background processing happens when the screen is off, but the phone is still running background tasks such as checking for email, and so on.

6.4 Limitations and Opportunities

NEAT was designed with two goals in mind: (i) to gather more accurate energy measurements in a portable manner (hardware component), and (ii) to uncover cases in which accurate high-speed, in situ measurements are absolutely necessary to solve energy problems (software component). Both of these goals, in particular the second one, need further experiments to validate the indispensability of NEAT. Regarding the first goal, Section 6.1 showed that difficult-to-model components such as OLEDs require an external measurement unit. The need for high-speed, in situ measurement is however not equally strong, and existing model-based solutions, such as [16,17], could provide sufficient accuracy (assuming easy-to-model components). This is because most power spikes occur on top of a constant level of power consumption (cf. Figures 15(a) and 18(a)), and hence, lower sampling rates may only miss a small area of energy usage. Regarding the second goal, even though the short-lived events shown in Section 6.2 and Section 6.3 (Sampling Rate) were obtained via in situ traces, similar results could have been obtained in a lab. Also, note that we did not offer direct evidence that the application energy usage obtained from NEAT’s high-speed measurement is better than that obtained by low-speed methods such as [16,17].

Opportunities for future work. To overcome the limitations described in the paragraph above, our traces could be under-sampled to simulate the 100 Hz sampling rate of SoA model-based methods. This would allow a precise quantification of the extra accuracy provided by NEAT. The traces could also be explored in more detail to identify high-speed events that occur only in the wild, which would strengthen the case for accurate on-board monitor in smartphones. These and other potential evaluations will be open for the community to investigate, since our traces will be publicly available at the project’s website (http://code.google.com/p/neat-power-toolkit/).

Overall, in spite of not uncovering cases in which NEAT’s unique capabilities are absolutely necessary, our studies demonstrate that NEAT is as accurate as current high-speed power meters and can be deployed in a relatively unobtrusive manner. Therefore, we hope that NEAT will enable the community to take a much closer look into the energy consumption of mobile devices under realistic usage.

Figure 17. Power consumption broken down into suspend, background processing, and various apps.
7 Related Work

In Section 2 we briefly discussed the background work that inspired the design and development of NEAT. In this section we expand on some of the alternative methods that have been proposed by the research community.

Energy modeling. Battery monitoring units (BMU) are included by default in smartphones and are by far the easier method of measuring power consumption. Some modern BMUs have a good resolution (in the order of a few hundred µA) and a high accuracy (few percent points of error). The main limitation of these methods is the low update rate which misses short-lived events, which in turn leads to a low aggregated accuracy and the inability to detect the power consumption of some tasks and components.

This low update rate has motivated the research community to increase the precision of energy profiling via energy-models. Several notable contributions have been made in this domain, and to some extent, NEAT follows the same general principles defined by this first generation of models where: power measurements are collected with external equipment, system events are collected via software, and then, these two pieces of information are combined to provide insights about the energy consumption of the device. In one of the first studies following this research line, Flinn and Satyanarayanan propose a general trace-based modelling technique to map program structure to energy consumption in laptops [9]. Carrol and Heiser used a simple linear model to estimate the battery lifetime of smartphones for a diverse set of scenarios [5]. Pathak et al. enhanced external measurements with a finite state model to develop eprof [17], a tool with an update rate of 20 Hz that allows a fine-grained analysis of energy consumption in apps.

While the above mentioned methods are a big leap forward on energy analysis, the overhead of performing external measurements is high and needs to be repeated for each individual platform. This limitation motivated a new-generation of self-learning models. PowerBooster [22] uses the discharge curve of the phone’s battery plus a software training phase to model the consumption of six energy-hungry components (GPS, WiFi, GSM, etc). Without relying on external measurement equipment (to bootstrap the model), the authors report a power estimation error under 3%. On a similar line of work, Sesame [7] enhanced the update rate to 100 Hz obtaining an accuracy of 88%. At 1 Hz the authors report an accuracy of 95%.

As can be seen, the improvements achieved with model-based approaches are significant, and compared to our work, they have the advantage of not requiring extra hardware. These methods have, however, an important limitation: they can only model what can be observed at the software level, but as the complexity of smartphones increases so does the number of hidden states that can not be modelled. This limitation can lead to several undesirable outcomes. First, high-frequency power spikes such as those shown in Figure 18(a) would be lost because they are extremely short lived. Second, high-rate changing components such as RAM and OLED displays can not be captured either (as mentioned in [12,17]), and the popular linear regression methods used in most studies are not accurate enough to cope with modern multicores systems [13]. Third, measuring the power consumption of the 3G modem would be difficult because the state transitions of the modem (RAT switches) are not available when the CPU is in suspend mode. Furthermore, buildings models, like those in [7], require a type of built-in current sensors that are not present in all phones. Considering the above described scenario, our work aims at bridging the gap between model-based approaches and the factual trace obtained with more sophisticated (but static) hardware tools.

Portable Power Meters. External power meters such as the popular Monsoon power meter [15] limit mobility. To overcome this problem, some efforts have proposed more portable meters. BattOr [10,20] is an Arduino-based power meter that attaches to the back of a phone, and can measure current independently and write it to a micro-SD card. Unfortunately the Atmega328 microcontroller it is built around has an ADC with only 10 bits of resolution, and the design does not include an op-amp to boost the voltage difference to take full advantage of this range, leading to poor data quality. Another problem is that BattOr is powered by the phone battery itself rather than its own power source, which means that it influences the very signal it is trying to measure. The authors claim that this effect is small, but cite a power consumption of up to 181 mW, which we consider to be non-trivial. The power meter proposed in [11] improves on this design by using an AD8210 shunt monitoring IC to get the most out of the Atmega328’s ADCs, and is powered
by its own battery. Further, it has the capability of monitoring
three devices concurrently. But, because it is built from
standard Arduino parts and shields, it is rather bulky, and still
uses relatively coarse 10-bit measurements. Table 1 shows
how the NEAT power meter compares the aforementioned
solutions. Overall, we believe that NEAT improves upon the
existing mobile meters by (i) achieving better resolution and
sampling rate, ii) using an external battery, so as not to in-
fluence the measured signal, and iii) accepting an additional
input for synchronization with the phone, which allows for
post-facto merging of power- and event traces.

8 Conclusions
The aim of our study is to provide a different alterna-
tive for the analysis of energy consumption in smartphones.
On the one hand, sophisticated hardware tools do not allow
mobility, but provide high precision. On the other hand,
model-based tools are portable, but lack accuracy and res-
olution. With NEAT we trade off a bit of precision (com-
pared to power meters) and a bit of portability and usability
(compared to model-based approaches) to cover a space
that has been yet unexplored, that of collecting and analyz-
ing highly accurate power traces while having the user in the
loop. NEAT allowed us to (i) highlight the importance of
having accurate meters to measure the energy consumption
of complex pieces of hardware such as OLED panels, (ii)
identify kernel bugs that would have been difficult to solve
without fine-grained traces, and (iii) provide a deeper insight
into the energy consumption of end-users due to the high
sampling rate and resolution of NEAT.

It is important to remark that NEAT is not aimed at re-
placing current methods. For non-mobile scenarios requir-
ing the highest precision, Moonson-like devices –possibly
coupled with our software tools– could be the best choice.
For crowdsourcing scenarios, where having access to a large
number of users is more important than the precision of the
data, model-based methods are arguably a better approach.
NEAT is mainly aimed at scenarios where the complex inter-
play of hardware, software, usage patterns, and environmen-
tal factors play a central role in the energy consumption
of the phone and a very detailed view is required to under-
stand the underlying dynamics or to solve potential problems.

9 Acknowledgements
We like to thank our reviewers of the paper for their ex-
tensive feedback. We also like to thank Winelis Kavelaars
and Frank van den Hoef from SOWNet Technologies for their
support in manufacturing the NEAT board. Finally, we thank
our colleagues Yunus Durmus, Michael Golinski, Qingzhi
Liu, Andreas Loukas, and Przemek Pawelcza for becoming
early adopters of NEAT-enabled smartphones and participat-
ing in our data collection campaign.

10 References
issues/detail?id=63427.
issues/detail?id=63432.
kodak/.
wi-fi scanning for energy-efficient localization. In IEEE PerCom ’14.
in a smartphone. In USENIX’10.
energy consumed in smartphone display applications? In ACM Hot-
Mobile ’13.
energy modeling for battery-powered mobile systems. In ACM Mo-
biSys’11.
[8] Palaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D.,
Govindan, R., and Estrin, D. Diversity in smartphone usage. In
ACM MobiSys ’10.
for profiling the energy usage of mobile applications. In IEEE WMCSA
'99.
Y., and Li, M. E3: Energy-efficient engine for frame rate adaptation
DIY power monitor to compare mobile energy consumption in situ.
In MobileHCI’13.
[12] Jung, W., Kang, C., Yoon, C., Kim, D., and Cha, H. Devs-
cope: A nonintrusive and online power analysis tool for smartphone
hardware components. In IEEE/ACM/IFIP CODES+ISSS’12.
[13] McCullough, J., Agarwal, Y., Chandrashekar, J., Kup-
puswamy, S., Snoeren, A., and Gupta, R. Evaluating the effec-
tiveness of model-based power characterization. In USENIX’11.
[14] Mittal, R., Kansal, A., and Chandra, R. Empowering devel-
opers to estimate app energy consumption. In ACM MobiCom’12.
PowerMonitor/.
[16] Pathak, A., Hu, Y., and Zhang, M. Where is the energy spent
inside my app?: Fine grained energy accounting on smartphones with
my phone awake?: Characterizing and detecting no-sleep energy
bugs in smartphone apps. In ACM MobiSys’12.
en/docs/Rhino.
[19] Schulman, A., Navda, V., Ramjee, R., Spring, N., Deshpande,
P., Grunewald, C., Jain, K., and Padmanabhan, V. N. Bartender: A practical approach to energy-aware cellular data
Phone power monitoring with BattOr. In ACM MobiCom’11.
[21] Yoon, C., Kim, D., Jung, W., Kang, C., and Cha, H. App-
scope: Application energy metering framework for android smart-
[22] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R., Mao,
Z., and Yang, L. Accurate online power estimation and automatic
battery behavior based power model generation for smartphones. In
IEEE/ACM/IFIP CODES+ISSS’10.
[23] Zhou, R., and Xing, G. Nemo: A high-fidelity noninvasive power