Calculating Source Line Level Energy Information for Android Applications

Ding Li, Shuai Hao, William G.J. Halfond, Ramesh Govindan
Department of Computer Science
University of Southern California
Los Angeles, California, USA
{dingli, shuaihao, halfond, ramesh}@usc.edu

ABSTRACT

The popularity of mobile apps continues to grow as developers take advantage of the sensors and data available on mobile devices. However, the increased functionality comes with a higher energy cost, which can cause a problem for users on battery constrained mobile devices. To improve the energy consumption of mobile apps, developers need detailed information about the energy consumption of their applications. Existing techniques have drawbacks that limit their usefulness or provide information at too high of a level of granularity, such as components or methods. Our approach is able to calculate source line level energy consumption information. It does this by combining hardware-based power measurements with program analysis and statistical modeling. Our empirical evaluation of the approach shows that it is fast and accurate.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Measurement, Performance

Keywords

Energy measurement, Android app, Source line level

1. INTRODUCTION

The popularity of mobile apps continues to increase. This popularity is driven, in part, by the innovative ways in which app developers combine sensors and data to provide users with useful and novel functionality. However, a problem for developers and users alike is that these apps require a large amount of energy and the mobile devices on which the apps run are constrained by limited battery power. This has led to a tension between adding new features that will attract users, but consume more energy, and minimizing energy costs at the risk of reducing functionality. The balance has proven hard to attain and it is quite common to see many complaints related to energy consumption in app marketplace reviews.

Research advances in battery, hardware, and operating system design have, to some extent, improved the energy consumption of mobile devices. However, in spite of these optimizations, a poorly-coded app can be inefficient and perform numerous unnecessary and costly operations. It is difficult for developers to determine if their app is coded inefficiently with respect to energy consumption. Although developers can consult well-known best practices, these do not represent an objective way to improve an app's implementation. An additional problem is that these guidelines do not provide application specific guidance, therefore developers lack insight into how energy is consumed by their apps.

To assist developers, researchers have developed a range of techniques to provide energy consumption information. Several techniques use runtime monitoring to track key operating system parameters and provide estimates to developers [8, 24, 39]. However, the level of granularity of these techniques is either at the component or method level, which is helpful, but does not provide information at a low enough level of granularity to guide developer changes. For example, method level information cannot help developers distinguish between two paths within a method that have different energy consumption. Developers have often used CPU time as a proxy for energy. However, this is not an accurate approximation because mobile devices scale their voltage dynamically and interact with multiple hardware components that have varied energy consumption patterns, such as GPS, WiFi, and cameras [12]. Techniques for estimating energy consumption (as opposed to measuring) could also be used, but have drawbacks: Cycle-accurate simulators [22] often run several thousand times slower than actual hardware, and program analysis based estimation techniques [12, 27] require carefully fine-tuned software environment profiles.

Current research has not been able to provide developers with techniques that can use energy measurements to provide source line level energy consumption information. As we discuss in more detail in Section 2, there are numerous practical and conceptual challenges in providing this information. The most straightforward technique, using a power meter to take direct measurements of an app while it is running, is not practical. Current power meters are unable to measure and record fast enough to isolate individual source lines. Even if this problem could be resolved, such measurements would not be accurate unless they also dealt with issues, such as thread switching and garbage collection, that can make it difficult to determine the implementation structures that should be attributed with the measured energy cost. As we discuss in Section 3.2.2, these types of events can consume a significant amount of energy that could distort the energy attributed to the application.

In this paper, we present a new approach that provides developers with source line level energy information. To address the numerous inherent challenges in achieving this result, we have devised

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '13, July 15-20, 2013, Lugano, Switzerland
Copyright 2013 ACM 978-1-4503-2159-4/13/07 ...$15.00.

an approach that combines hardware-based energy measurements with program analysis and statistical modeling techniques. At a high-level, the basic intuition is as follows: While measuring the energy consumption of a smartphone, the approach uses efficient path profiling to identify which parts of the application are executing and correlates these paths with the measured energy. Then, the approach statically analyzes the paths to identify and adjust for high-energy events, such as thread switching, before applying robust regression analysis to calculate each source line's energy consumption. Finally, the approach presents developers with a graphical representation of the energy consumption by overlaying the calculated energy with the source code of the application.

We also performed an empirical evaluation of our approach to measure its accuracy and the time needed to perform the analysis. Our approach was able to accurately calculate energy. For a set of API invocations, we found that the calculated energy values were within 10% of the ground truth measurements. The statistical models also matched the measured data very closely with a high R^2 average of 0.93 and a low accumulated error rate. Importantly, in our experiments, the approach was also able to detect the influence of high-energy events, such as thread switching and garbage collection, with 100% accuracy. The approach was also fast, it could calculate source line level energy measurements for each of our subject applications in less than three minutes. Overall, the results of the evaluation were positive and indicate that our approach is an effective and practical way to provide developers with source line level energy information.

The remainder of this paper is organized as follows: In Section 2 we discuss several of the significant challenges that shaped the design of our approach. We present the approach itself in Section 3. The evaluation of the approach is described in Section 4. Finally, we discuss related work and conclude in Sections 5 and 6.

2. BACKGROUND AND MOTIVATION

In this section we discuss several challenges to measuring and calculating source line energy consumption. We break the challenges into three broad categories, hardware, runtime system, and software level, and explain how they preclude a straightforward solution for calculating source level energy consumption. To illustrate these challenges, we make use of the code shown as Programs 1 and 2, which are excerpts from the open-source Google Authenticator project that handles synchronizing a system clock. In the example, method `getNetworkTime` (Program 1, line 1) retrieves the time by sending an HTTP request to a timeserver (line 7) and then parsing the response packet (lines 16, 20, 23, 26). Method `runBackgroundSync` (Program 2, line 2) calls the `getNetworkTime` (line 6) and then starts a background thread to handle errors (line 10).

2.1 Hardware Limitations

The primary hardware obstacle to directly measuring the energy consumption of source lines is the difference in the speed at which instructions execute and hardware devices can perform energy measurements. On modern processors, individual instructions will execute at a rate of several million per second. At best, power meters can sample electrical power draw at several tens of KHz [16], which means that each sample will include the power consumption of hundreds, perhaps thousands, of instructions. For example, it is possible for Program 2 to execute completely in the time that transpires between two consecutive power samples by a relatively fast power meter. There are reasons to believe that this order of magnitude disparity will likely persist, since the bottleneck in high-frequency power sampling is the storage system, which cannot save

```
1   public long getNetworkTime() throws IOException {
2     HttpHead request = new HttpHead(URL);
3     Log.i(LOG_TAG, "Sending request to "
4           + request.getURI());
5     HttpResponse httpResponse;
6     try {
7       httpResponse = mHttpClient.execute(request);
8     } catch (ClientProtocolException e) {
9       throw new IOException(String.valueOf(e));
10    } catch (IOException e) {
11      throw new IOException("Failed due " +
12                  "to connectivity issues: " + e);
13    }
14
15    try {
16      Header dateHeader =
17          httpResponse.getLastHeader("Date");
18      Log.i(LOG_TAG, "Received response with "
19            +"Date header: " + dateHeader);
20      if (dateHeader == null) {
21        throw new IOException("No Date header");
22      }
23      String dateHeaderValue =
24                  dateHeader.getValue();
25      try {
26        Date networkDate =
27                  DateUtils.parseDate(dateHeaderValue);
28        return networkDate.getTime();
29      } catch (DateParseException e) {
30          throw new IOException(
31                  "Invalid Date header format ");
32        }
33      } finally {
34          ......
35      }
36    }
```

Program 1: Google Authenticator Method 1

power samples at the same frequency as the power meter can generate them [14, 15, 20], and there is no evidence to show that this gap will be closed in the near future. Therefore, a challenge for our approach is to reconcile power measurements samples and instruction execution, even though there is a several orders of magnitude difference in their frequency.

2.2 Runtime System Challenges

The runtime system of an Android smartphone includes both the Android operating system and the Dalvik Virtual Machine, the platform on which marketplace apps are run. The runtime system implements several types of behaviors that affect energy consumption of an app, thread switching, garbage collection, and tail energy. However, the details of the duration, frequency, and timing of these events is, by design, hidden from the app. This makes it difficult to correctly attribute energy at the source level. Although it would be straightforward to modify the runtime systems to track these events, this would introduce considerable overhead and reduce the portability of the approach, as it would be necessary to provide custom runtime systems for each smartphone platform. Therefore, one challenge for our approach is to account for these events with information that is available at the app layer.

Thread switching causes several problems for calculating source line level energy. The first problem is that any energy measurements for a given time period may also include energy for threads that are not related to the application (*i.e.,*, operating system processes). For example, at any point in the execution of the thread in `runBackgroundSync` or `getNetworkTime`, the OS could swap in a thread from another app or the OS. Any power samples during this time would likely include instructions from both the original and swapped-in thread. The second problem is that thread

```
1   private void
2     runBackgroundSync(Executor callbackExecutor){
3       long networkTimeMillis;
4         try {
5           networkTimeMillis =
6             mNetworkTimeProvider.getNetworkTime();
7         } catch (IOException e) {
8           Log.w(LOG_TAG, "Failed to obtain network " +
9                   "time due to connectivity issues");
10          callbackExecutor.execute(new Runnable() {
11            @Override
12            public void run() {
13              finish(Result.ERROR_CONNECTIVITY_ISSUE);
14            }
15          });
16          return;
17        }
```

Program 2: Google Authenticator Method 2

switching itself introduces an energy overhead. For example, Line 10 of Program 2 will incur a significant energy overhead for the context switch that occurs when a new thread is started.

Periodically, the Android operating system will perform garbage collection during the execution of an application. Typically, applications do not have control over garbage collection, it is managed by the operating system and can occur anytime and within any method. Although the Dalvik virtual machine logs garbage collection, it timestamps the events via a millisecond based clock, which does not provide enough precision to determine in which path they occur (paths are tracked with nanosecond timestamps.) As we note in Section 3.2.2, garbage collection incurs a significant energy cost and has the potential to significantly distort energy measurements for an application. For example, consider if garbage collection were to occur at line 20 of Program 1, which is compiled as a simple jump on equality (`ifeq`), or line 6 of Program 2, which is a relatively expensive method invocation. The extremely high cost of the garbage collection would dwarf the cost of both instructions making them appear to have a nearly identical high energy cost. Therefore, an approach must be able to identify and properly account for garbage collection while calculating the source line level energy consumption of an application.

Smartphones exhibit tail energy usage, where certain hardware components are optimistically kept active by the operating system, even during idle periods, to enable subsequent invocations to amortize startup energy costs. Tail energy manifests itself in two different ways. The first way is when an invocation to an API accesses a hardware component and then returns. Even when no other invocations access the component, the component will remain active for a time period T_{tail} and consume energy E_{tail}. An example of this kind of invocation is at line 7 of Program 1. The call to the network will cause the radio to remain on for a period of time, even after the request is finished, and consume energy during this time. The second way is when two invocations access the component in sequence and the time interval between them is less than T_{tail}. Here, only a portion of E_{tail} will be consumed. The E_{tail} for certain devices can be quite high and a naive approach to measurement might attribute the tail energy cost to subsequently executed instructions. Referring back to the example invocation, the naive approach could mistakenly attribute part of the invocation's tail energy to its successors, lines 16, 18, and 20, (depending on the length of its T_{tail}) instead of to line 7. To be accurate, an approach must recognize when tail energy occurs and attribute it to the source invocations that started the component and kept it at a non-idle state.

2.3 Software-Level Challenges

Source level calculations require precise information about an app's execution. In particular, for each time period represented by a power sample, it is necessary to know which instructions were executed, their frequency, and their ordering — essentially, path information. The primary challenge is that obtaining this type of information at the software level is generally expensive and intrusive. For example, inserting instrumentation into an app to isolate and measure individual instructions can lead to high overhead. Even optimizations, such as only instrumenting each basic block, would still produce a high amount of instrumentation. Other approaches, such as instruction counting, do not allow developers to know which instances of the instructions were executed.

It is also necessary to have precise information about API invocations. During execution, smartphone applications generally invoke library functions and APIs to access hardware components, such as GPS, network, and WiFi. For example, at line 7 of Program 1, `getNetworkTime()` sends an HTTP request to Google by invoking `HttpClient.execute()`. This invocation consumes a relatively large amount of energy due to its use of the network. In contrast, a call to `dataheader.getValue` at line 24 of Program 1, incurs a relatively small constant cost, since it is simply returning the value of an HTTP header. This wide range of invocation behavior causes problems for techniques that naively apply statistical sampling or averaging-based approaches because, unlike normal instructions, an invocation's energy cost can vary based on its target and the data provided for its arguments. Furthermore, it is not feasible to precompute the cost of API calls, since that requires sampling every point in each API calls input space to determine its energy consumption.

3. APPROACH

The goal of our approach is to provide source line level energy information for smartphone applications. An overview of our approach is shown in Figure 1. From a high-level, our approach has two phases, Runtime Measurement and Offline Analysis. The inputs to the Runtime Measurement phase are the application under analysis (AUA) and a set of use cases for which the tester wants to obtain energy measurements. The App Instrumenter uses an efficient path profiling technique to guide the insertion of probes into the AUA that will capture timestamps and path traversal information. The tester executes the instrumented AUA on the Power Measurement Platform. This causes the instrumentation to record path information while the Power Measurement Platform records power samples. The path and power samples are the inputs for the Offline Analysis phase. The Path Adjuster performs a static analysis of the paths and makes modifications to the power samples to account for the high-energy events. Then the Analyzer performs the regression analysis in order to calculate each source line's energy consumption. If the Analyzer finds there are not enough data points to solve the regression analysis, then the process returns to the Runtime Measurement phase so the developer can further execute the application. If this is not possible, then the Analyzer performs approximations that we discuss in Section 3.2.2. Finally, the Annotator uses the calculations to create graphical overlays of the measurements on the source code for display in an integrated development environment. We explain the approach in more detail in the remainder of this section.

3.1 Runtime Measurement Phase

During the Runtime Measurement phase, information about the paths executed in the AUA and power measurements are generated by the approach. To do this, the App Instrumenter inserts probes

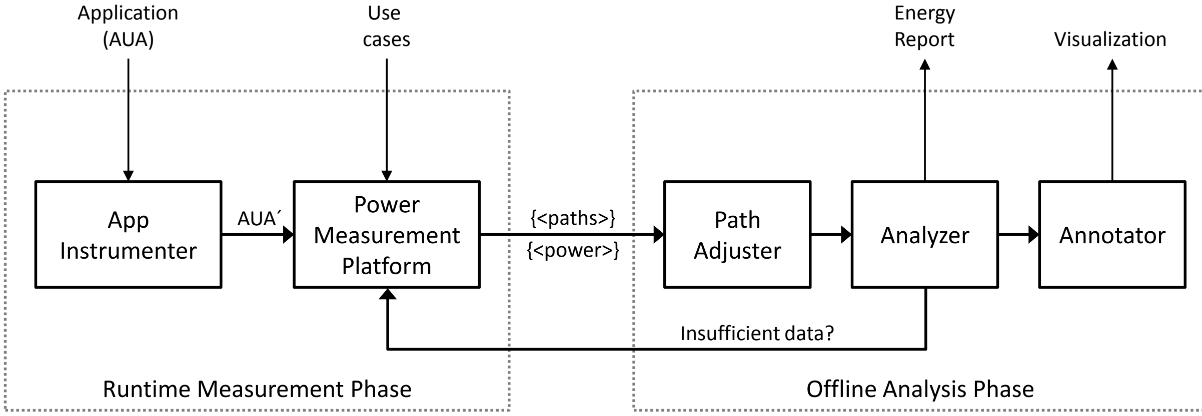

Figure 1: Overview of the approach

into the AUA to record the information, and then the instrumented AUA is executed by the developer while power samples are collected by the Power Measurement Platform. The output of the Runtime Measurement phase is set of timestamped paths executed by the AUA and power measurements.

Instrumentation of the AUA: The instrumentation collects information about the paths executed by the developer. Namely, which paths are traversed, their frequency, and timestamps of the path traversals and invocation of certain APIs. To record the path information, the App Instrumenter adapts a techniques for efficient path profiling proposed by Ball and Larus [5]. The approach first builds a control-flow graph (CFG) of each method in the AUA. Then each edge in the CFG is assigned a label so that each unique path in the CFG has a unique path ID. The approach then calculates a maximal spanning tree over the CFG and uses this to guide the minimal placement of instrumentation that will increment a path ID counter. By design, the approach can use a single counter to identify the path traversed in the method. We extended the Ball-Larus approach to handle nested method calls, concurrency, and exceptions. The App Instrumenter also inserts a probe at the method entry to initialize the method's path counter, record a timestamp of when the path traversal began, and the current thread ID. At each exit point of the method, another probe records the value of the path ID counter, and another timestamp. After execution is finished, this information allows the approach to generate a set of *path tuples* of the form $\langle thread_id, path_id, enter_time, exit_time\rangle$ where $thread_id$ identifies current thread, $path_id$ is the traversed ID for paths, and $enter_time$ and $exit_time$ are the time stamps that indicate when a path starts and ends.

The App Instrumenter also inserts probes to obtain timestamps before and after the invocation of certain APIs. As we explain in Section 3.2.1, this information is used to allocate tail energy and isolate the energy cost of certain APIs for which it is not possible to model using linear regression (i.e., methods that have non constant energy consumption.) For each of the invocations, the instrumentation generates an *invocation tuple* of the form $\langle method_id, enter_time, exit_time\rangle$, where $method_id$ identifies the invoking method, and $enter_time$ and $exit_time$ are timestamps before invoking the method and after the method has returned.

Execution of the Instrumented AUA: To generate the path and invocation tuples, the instrumented AUA is executed on the Power Measurement Platform. The Power Measurement Platform is based on the LEAP node [25]. The LEAP is an x86 platform based on an ATOM N550 processor that runs Android 3.2. Each component in the LEAP (*e.g.,* WiFi, GPS, memory, and CPU) is connected to an analog to digital converter (DAQ) that samples current draw at 10KHz. The LEAP also provides Android applications with the ability to trigger a synchronization signal. This allows the approach to synchronize the samples with the paths' timestamps and avoid inaccuracy due to clock skew. Each of the uses cases is executed by the developer on the AUA while it is running on the LEAP. Note that all of the measurements are recorded in hardware external to the Android smartphone components, so the measurement process does not introduce any interference or execution overhead.

3.2 Offline Analysis Phase

In the Offline Analysis phase, our approach analyzes the tuples and power samples generated in the Runtime Measurement phase to produce a mapping of energy to source lines. There are three parts to this phase. In the first part, the Path Adjuster statically examines each traversed path in the CFG and adjusts the corresponding energy measurements to account for special API invocations, tail energy, and interleaving threads. The adjusted energy measurements and paths are the inputs to the second part, the Analyzer, which uses robust regression techniques to calculate the cost of each source line and identify paths along which garbage collection and thread context switches occurred. The Analyzer also determines if the testing process created enough data points to perform the linear regression and either reduces the grouping of variables to be solved in the regression or directs the tester to repeat the test cases to provide more data points. Finally, in the third part, the Annotator creates a graphic representation of the energy measurements and overlays this with the source code. Note that in the rest of the paper, we describe our approach in terms of bytecode instructions, but it is straightforward to convert the bytecode-level information to source-level using compiler provided debugging information.

3.2.1 Adjustment to Path Energy Samples

Before beginning the analysis, the Path Adjuster first reconstructs the paths traversed during execution of the AUA. The instructions executed in a path can be identified using the path ID and the CFG of the method containing the path, as described by Ball and Larus [5]. Once the paths have been reconstructed, the Path Adjuster calculates the energy total for the path by summing the measurements reported during the path's time of execution. The Path Adjuster can identify the corresponding path and power samples due to the synchronized timestamps. Then, the Path Adjuster per-

forms a static analysis of each path in order to adjust certain API invocations due to non-constant and too-short API invocations, tail energy, and thread interleaving. The Path Adjuster generates a set of paths with the adjusted invocations removed from the paths and their corresponding energy removed from the power measurements.

API Invocations: Certain API invocations have a non-constant energy cost associated with their execution. Therefore, it is not possible to calculate their energy cost using the robust linear regression techniques described in Section 3.2.2. To address this problem, our approach uses the invocation tuples to identify the time periods when these invocations are executing and calculates the invocations' energy cost by summing the power measurements taken during that time.

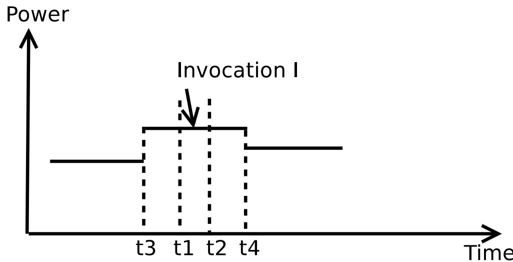

Figure 2: Using longer execution windows for calculating the energy of invocations with a short execution time.

In most cases, the execution time of an invocation is long enough that the approach is able to get accurate energy measurements (i.e., the execution time is longer than several sampling periods.) However, in some cases the execution time of the invocation is too brief (e.g., the execution time is shorter than a sampling period.) For these invocations, the approach identifies an execution time period that includes the too short execution and for which the approach does have enough power samples to accurately the energy consumption. Then it calculates the ratio of the original execution time versus the larger execution time and multiples that against the energy total for the larger execution time. To illustrate, consider the example shown in Figure 2. In this figure, the horizontal bars indicate the energy sampling interval. For example, the LEAP will sample the power at $t3$ and then again at $t4$. If an invocation I executes from time $t1$ until time $t2$, then there are no power samples to be summed in order to find the energy consumed by I (E_I). When this occurs, the Path Adjuster finds the next largest execution window for which it has sufficient energy samples and uses this window to calculate E_I. In the case of the example, the Path Adjuster calculates E_I as shown in Equation 1, where $E_{a,b}$ denotes the energy consumption measured by the Power Measurement Platform during interval $[a, b]$.

$$\frac{t2 - t1}{t4 - t3} E_{t3,t4} \tag{1}$$

Note that this only approximates E_I. We have found that for functions whose execution time is so short, this is a reasonable approximation. These functions consume a very small portion of the overall energy expended at runtime, on average about 6% of the total API energy cost. For the general case, where the execution time is of sufficient length, our evaluation shows that we are able to accurately measure most functions to within 9% of their measured ground truth cost.

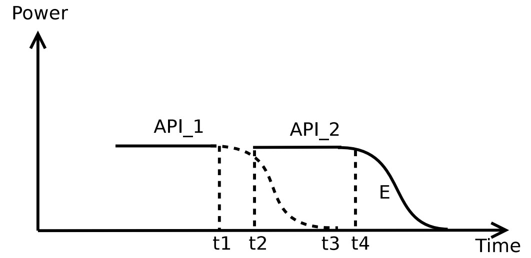

Figure 3: API invocations with tail energy

Tail Energy: As explained in Section 2, tail energy occurs when the operating system keeps certain hardware components active, even during idle periods, to enable subsequent invocations to amortize startup energy costs. The result of this behavior is that the energy measurements for the time period following the component access will be higher. Our approach adjusts the path energy total so that the tail energy is attributed to the invocations that interact with the hardware component. We assume the availability of tail energy models, which are generally provided by either component manufacturers or power researchers [24]. The model specifies the energy consumption of the component after an invocation (E_{tail}) and for how long the device driver maintains this state (T_{tail}). To calculate the adjustment, the Path Adjuster examines the reconstructed paths to identify the sequence A_D of invocations to methods that cause tail energy for each device D of the smartphone. For each such invocation $a_i \in A_D$, the adjuster compares the timestamp of a_i against the timestamp of a_{i+1}. If the difference is greater than T_{tail} then all of E_{tail} is attributed to a_i. If the difference is less than T_{tail} then only a fraction of E_{tail} is expended before a_{i+1} occurs and should be attributed to a_i. The fraction is calculated as shown in Equation 2, where T_S returns the starting timestamp of an invocation and T_E returns the ending timestamp of an invocation. Note that the invocation timestamps are known due to the invocation tuples collected during the Runtime Measurement phase.

$$\frac{T_S(a_{i+1}) - T_E(a_i)}{T_{tail}} E_{tail} \tag{2}$$

To illustrate these two scenarios, consider the two invocations shown in Figure 3. Both of these, API_1 and API_2, access the same device in sequence. Their tail energy consumptions are shown as curved lines extending after the end of the invocations. To calculate the tail energy associated with API_1, note that $t2$, the start of the invocation to API_2, occurs before the T_{tail} time has transpired. Therefore the tail energy assigned to API_1 is calculated as $\frac{t2-t1}{t3-t1} * E_{tail}$. When the invocation to API_2 returns at $t4$, there is no other API accessing the same external device, so we assign all of E_{tail}, to API_2.

Thread Context-Switching During Invocations: A potential problem that arises with the way the approach attributes energy to API invocations is that it is possible for the thread containing the invocation be switched out for another thread from the same application. For example, this can happen because the invocation must wait for a shared resource or another thread has higher priority. The problem is that energy consumption by the other threads will then also be measured in the corresponding power samples. As we discussed earlier, this situation can be detected by modifying the operating system to accurately track thread scheduling. However, this would make the approach less portable, so we have devised a software-level technique to address this problem.

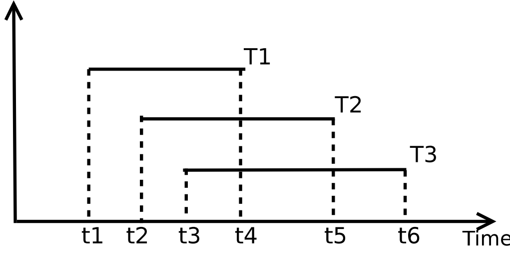

Figure 4: Concurrent threads during an API invocation.

The Path Adjuster determines the number of threads that were executing while the invocation was executing. This is done by examining the starting and ending timestamps of each path tuple to see which ones were active between the invocation's timestamps. Then, the Path Adjuster evenly allocates the energy among the concurrent threads, assigning each the energy of $\frac{1}{N}$, where N is the number of concurrent threads. To illustrate, consider the example shown in Figure 4. $T1$ is the original thread and contains an API invocation at time $t1$. Threads $T2$ and $T3$ run while $T1$ is performing the invocation. $T1$ and $T2$ are concurrent in time interval $[t2, t3]$ and all three are concurrent in $[t3, t4]$. Therefore, the energy of $T1$ will be $E_{t1,t2} + \frac{1}{2}E_{t2,t3} + \frac{1}{3}E_{t3,t4}$. This is similar to the way prior approaches have handled concurrent thread energy [24].

3.2.2 Calculating Source Line Energy Values

The second part of the Offline Analysis phase calculates the energy consumption of each source line. The input to the Analyzer is the set of adjusted paths and energy samples produced by the the Path Adjuster. At this point, adjustments for all of the API invocations have been made, but the paths could still be influenced by the occurrence of garbage collection and thread switching during times when there is no API invocation. As we discussed in Section 2, the energy costs associated with events, such as garbage collection and thread switching, can skew the regression analysis, but it is difficult to identify when they occur. Our insight is that characteristics of these events allow us to define them as statistical outliers and therefore the Analyzer can employ Robust Linear Regression techniques to calculate each source lines' energy consumption and mitigate the influence of these high-energy events. In the rest of this section, we first explain the regression technique and then discuss the insight that allows us to define garbage collection and thread switching as statistical outliers.

Robust Linear Regression Analysis

The Analyzer uses linear regression analysis to calculate each instruction's energy consumption. We expect linear regression to work well in this situation because prior work has found that the cost of bytecodes will be constant given a particular hardware environment [12, 29] and the Path Adjuster has removed the non-linear cost invocations.

To perform the analysis, the Analyzer sets up the equations $\boldsymbol{E} = \boldsymbol{mX}$, where $\boldsymbol{E}$ is the adjusted power measurements, $\boldsymbol{X}$ represents the path traversals with each row representing a frequency vector of bytecodes present in the measured path. (Note that the bytecodes associated with the inserted instrumentation are included in this matrix. Since the approach knows which paths were instrumented, the associated results are simply removed before visualizing or reporting the final source line level calculations.) Then the Analyzer solves for the coefficients $\boldsymbol{m}$ to determine the energy consumed by the instructions in the path segment. To solve the equations, the Analyzer employs Robust Linear Regression (RLR) analysis. In particular, the Analyzer uses RLR based on M-estimation [13], which does iterative regression analysis. It begins by solving a normal linear regression on the set of data points. Then at each iteration, it calculates the residuals and gives each data point a weight based on the standard deviations of the residuals. The regression analysis is repeated on the weighted data to generate a new model and the process repeats until the standard deviation of residuals does not change between iterations. For the power samples, RLR is preferable over the well-known ordinary least squares approach because it is more robust in the presence of outliers. In this case, our data sample has outliers, which are the paths whose energy measurements are influenced by garbage collection and thread switching.

More specifically, at each iteration, given the linear function $\vec{y} = X\vec{\theta} + \vec{u}$, the Analyzer solves Equation 3 and updates the standard deviation of residuals. For the weighting function (ψ), we use the well-known Tukey's Bisquare function [37], which is shown in Equation 4. The σ is the standard deviation of residuals in the last iteration. The value of k is constant and is set according to different use cases. In our experiments, we found that the average energy cost of garbage collection and thread switching is about 10 to 70 times the average standard deviation of the residuals in the first iteration; therefore, we select the median 40 of this range as k.

$$\sum_i \psi(y_i - \sum_k x_{ik}\theta_k)x_{ij} = 0 \tag{3}$$

$$\psi_k(x) = \begin{cases} x(k\sigma - x^2)^2 & -k\sigma < x < k\sigma \\ 0 & otherwise \end{cases} \tag{4}$$

The solution represents the energy cost of each instruction in the path. Taken together with the measured energy cost of the invocations, the Analyzer now has the energy cost for the entire path. The values for all of the paths are provided as input to the Annotator. Two special cases are discussed below.

The first special case is when it is not possible to solve for a path's instruction energy (i.e., $\boldsymbol{m}$). This happens when the number of unique bytecodes in a path is higher than the number of independent data points. Solving linear system of equations requires that there be at least as many independent data points as unknown variables. In this situation, the Analyzer recognizes that there are not enough data points and can take two actions, which are repeated until the equation is solvable. The first is that the tester is notified that the application should be executed more to generate additional data points. The additional executions do not need to exactly reproduce the initial executions, but should represent similar use cases to ensure a significant amount of path overlap. Since this is not always possible, the second possible action for the Analyzer is to group counts of similar bytecodes. For example, all variations of the `iconst` instruction. This gives fewer unknown variables for the system of equations. Note that in our experience, even moderate size marketplace apps were sufficiently complex that neither of these actions were required in our evaluation.

The second special case are paths identified as outliers, which contain external thread switching or garbage collection. The detection of these outliers is discussed more below. Currently, the Analyzer excludes these paths, which comprise about 1% of the total path count. Although it is desirable to include these paths, since excessive high-energy events could be a symptom of energy inefficient coding, there are two obstacles to this that we hope to address in future work. First, the path must be adjusted to remove the energy cost of the high-energy event. However, because the measured energy of these events is so high compared to the path energy, it is not clear how to accurately estimate and separate the

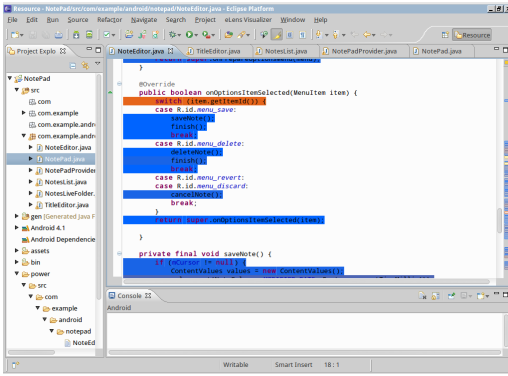

Figure 5: Visualization of the energy measurements.

event energy versus the path energy. Second, although the energy associated with these events is significant and of interest to developers, its not clear which instructions should be attributed with the cost. Currently, this number is tracked and reported as a separate total.

High-energy Events as Statistical Outliers

The energy overhead of switching to external threads and garbage collection is quite high. Our experiments show that these events range from 20,000 to 150,000 times the cost of a normal instruction. Yet they occur rarely during execution. Our insight is that the high energy cost and relative infrequency of these high energy events allows them to be detected as statistical outliers. In small scale experiments, we found that the energy cost of a path ranges from 0.007mJ to 0.631mJ; garbage collection from 20mJ to 81mJ; and thread switching overhead from 6.44mJ to 11.3mJ. This means that the energy cost of paths with garbage collection and thread switching will be significantly larger than normal paths — 10 to over 10,000 times larger. This difference in the range of values allows us to identify the events by defining them as outliers based on their energy consumption (*i.e.,* by setting an appropriate value of k in Tukey's Bisquare function.) Note that our approach detects outliers for a specific path and does not simply cluster all of the energy measurements. A per-path outlier detection is necessary since it is possible for a path to be long enough that it consumes more energy than garbage collection or thread switching. In the evaluation, we validate this approach to detecting garbage collection and thread switching by showing that our approach is able to detect all known occurrences of these types of events.

3.2.3 Visualizing the Energy Consumption

The Annotator presents a graphical representation of the source line level information. Developers can use this visualization to more readily understand the distribution of energy consumption across the different parts of their application. The Annotator was developed as part of our prior work [12], so it is not a research contribution of this paper. Nonetheless, we consider it an integral aspect of our approach that increases its usability, so we briefly summarize its function.

The Annotator is an Eclipse plugin that overlays power information onto an application's source code. A screenshot is shown in Figure 5. The visualization uses a SeeSoft [9] like graphical representation where different colors indicate the amount of energy consumed by source lines. The color for each source code line is obtained by ranking each source line according to the sum of its associated bytecode energy costs. The ranking is then mapped to a color within the spectrum. In our example, blue shows low energy consumption and red indicates a high level of energy consumption. In between values are different levels of red and blue (purple.)

Table 1: Subject applications

App	Application Information			
	C	M	BC	Description
BBC Reader	590	4,923	293,910	RSS reader for BBC news
Bubble Blaster II	932	6,060	398,437	Game to blast bubbles
Classic Alchemy	751	4,434	467,099	Educational game
Skyfire	684	3,976	274,196	Web-browser
Textgram	632	5,315	244,940	Text editor

4. EVALUATION

We evaluated two aspects of our approach, analysis time and accuracy. For the evaluation, we implemented our approach as a prototype tool, *vLens*, and designed several experiments to evaluate these two aspects. We considered two broad research questions:

RQ1: What amount of analysis time is incurred by *vLens*?

RQ2: How accurately does *vLens* calculate the energy consumption of the application?

4.1 Subject Applications

In the evaluation, we used a set of five application from the Google Play Market. Table 1 shows the number of classes (C), methods (M), and bytecodes (BC) for each application. All of the applications are written for the Dalvik Virtual Machine, do not use any native libraries, and can be translated to and from Java Virtual Machine bytecodes by the `dex2jar` tool. These applications represent real-world marketplace applications and implement a diverse range of functionality. For all of the experiments, we ran the applications using canonical usage scenarios for each application. For example, we played a game several times (Bubble Blaster II and Classic Alchemy), created and edited a text document (Textgram), read a news article via the reader (BBC Reader), and opened a web page (Skyfire).

4.2 Implementation

The *vLens* prototype is written in Java and works for Android applications written to run on the Dalvik Virtual Machine. We chose to implement for Android because its open nature made it easier to understand the OS inner workings. However, our approach is applicable for other platforms, such as Windows Phone and iOS, since it relies on energy measurements provided by external hardware, statistical techniques, and instrumentation, which is available on many platforms.

There are four main modules in the implementation: the Power Measurement Platform, App Instrumenter, Analyzer, and Annotator. For the *Power Measurement Platform* we utilized the LEAP power measurement device [25] described in Section 3.1. The *App Instrumenter* uses BCEL [4] to build intra-procedural control flow graphs for the efficient path profiling and insert the required instrumentation. We use `dex2jar` [1] to convert Dalvik bytecodes to Java bytecodes; then after instrumentation, we compile the classes back to Dalvik with the `dx` tool provided by the Android SDK. The *Analyzer* uses R [3] for the robust linear regression functions and Java code to perform the path adjustments. Finally, the *Annotator* is based on an Eclipse visualization plugin we built for prior work in energy estimation [12].

4.3 RQ1: Analysis Overhead

For the first research question, we consider three aspects of the

Table 2: Time and Accuracy

App	Timing Measurements $T_I(s)$	$T_A(s)$	$T_R(\%)$	Accuracy R^2	AEE (%)
BBC Reader	353	158	0.51	0.94	6.5
Bubble Blaster II	460	145	3.24	0.90	8.6
Classic Alchemy	873	128	8.77	0.93	3.4
Skyfire	277	97	1.12	0.99	4.8
Textgram	298	63	6.33	0.92	6.3

approach's analysis time. These are: (1) time to instrument each application (T_I), (2) time to perform the offline analysis (T_A), and (3) runtime overhead introduced by the instrumentation (T_R). Experiments to measure the first two were performed on a desktop platform containing an Intel i3@2.1Ghz with 2GB RAM and running Ubuntu 12.04. Overhead was measured on the LEAP platform.

To determine T_I, we measured the time to instrument each application with *vLens*. After running the application and collecting the path and invocation tuples, we determined T_A by measuring the time to analyze the tuples, perform path adjustments, and calculate each source line's energy using the regression analysis. The results of these measurements are shown, in seconds, in Table 2. The time to instrument ranged from five to twelve minutes, and the time to analyze ranged from one to just under three minutes. Most of the instrumentation time was due to the computational cost of building and analyzing the control flow graphs for each of the methods. The majority of the offline analysis cost was due to the IO overhead of reading all of the path tuples into memory so they could be converted to the path matrices. Further optimizations, such as caching path information during the offline analysis, could reduce this cost further. However, since *vLens* is intended to be an experimental prototype, we did not implement these improvements.

To determine the runtime overhead of the instrumentation (T_R) we could not take the straightforward approach of comparing an instrumented version of each application against the uninstrumented version. The reason for this is that a significant amount of application time is actually spent idle, waiting for user input or data. The normal user variation in entering this data masks the instrumentation overhead variation. Therefore we calculate the non-idle execution time of the instrumented application and then determine the percentage of that time that is caused by the instrumentation. A key insight to measuring non-idle time is that the Android operating system is event driven, so it idles waiting for user input after a method exits, and then when it receives input, comes out of the idle state and executes the event handler method. Therefore, we can calculate the non-idle time of the application by summing up all of the time during which an application path was being traversed. For example if path p_1 executed from time 1 to 3 and path p_2 executed from time 2 to 4, then we can calculate the non-idle time as 3 units. Note that this properly counts a time unit once regardless of how many threads are executing. Next, we profiled the instrumentation code that was inserted into the application and determined its execution time T_{Inst}. By examining the path tuples, we also know the execution frequency n of the instrumentation. The resulting calculation for T_R is shown in Equation 5. Here the denominator is the special time summation described above.

$$T_R = \frac{n * T_{Inst}}{\cup T_i} \quad (5)$$

The results of this calculation are shown in Table 2 as T_R. The overhead costs range from 0.51% to 8.77% with an average just under 4%. Overall, this is a low runtime overhead, representing about 0.15 seconds. Anecdotally, this amount of overhead did not cause a noticeable delay to the testers during execution.

4.4 RQ2: Measurement Accuracy

For the second research question, we consider the accuracy of the measurements calculated by *vLens*. The primary challenge in this evaluation is that there is no source line ground truth against which we can compare for accuracy. As discussed in Section 2, power samplers cannot measure at a frequency high enough to capture individual source lines. Therefore, we show accuracy of the approach in several other ways. First we examine the accuracy of the energy attributed to the APIs. Second, we use statistical tests to show that the regression analysis results can describe the overall energy consumption relationship accurately and that each path also accurately accounts for its energy. Lastly, we show that our technique for identifying high-energy events via statistical outlier analysis correctly identifies paths that contain these events.

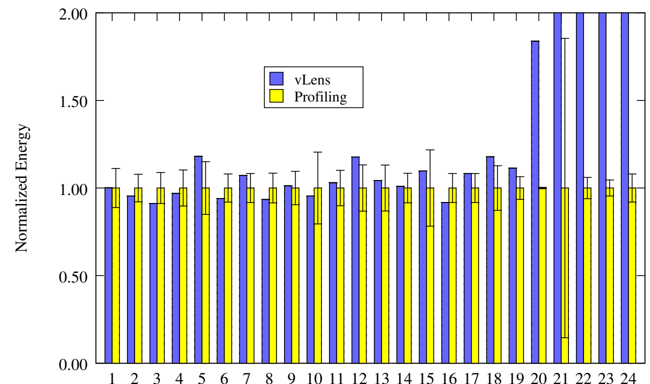

Figure 6: Comparison of API energy cost

4.4.1 Accuracy of the API Energy Measurements

To measure the accuracy of the API energy measurements, we compared their measured cost against a profiled cost. To obtain the measured cost, we ran the apps using *vLens* and extracted the cost of each invocation that was executed during the test runs. Altogether, there were invocations to over 3,722 unique APIs. From this group we focused on a group of twenty-four APIs whose invocations together comprised more than 70% of the total invocation related energy consumed by the five applications. For these twenty-four, we recorded the arguments and execution context of the invocations and then profiled their energy cost. The profiling was performed on the LEAP platform by executing each invocation 100 times and then measuring the energy consumed during the execution. We repeated this profiling five times and calculated the mean and standard deviation of the experiment. The profiled cost was then compared against the cost calculated by *vLens*. The result of this comparison are shown in Figure 6. Each of the APIs are listed along the X-axis and the two different costs are shown on the Y-axis, the values on the Y-axis are normalized to the cost of profiled cost. One standard deviation off of the profiling mean is shown with the additional horizontal lines.

Figure 6 shows that for nineteen of the twenty-four APIs, the *vLens* measured cost was within an average of 9% of the profiled costs and the gap was within one standard deviation in almost all cases. For the remaining five, the measured cost was off significantly. We investigated these APIs to determine what caused this high error rate. The first four APIs are synchronized, which means that they invoke `monitor_enter` and `monitor_exit`. This

made the execution time and energy consumption of their invocation vary widely as the acquisition and release of the synchronization lock was non-deterministic. The fifth API accessed an HTTP response code, and we found that the code was cached after the first call to the method, which meant significantly less computations had to be performed in subsequent invocations. Because the profiling results for these five were skewed by these types of behaviors, we consider the nineteen to be a more accurate reflection of the accuracy achieved by the *vLens* calculations.

4.4.2 Accuracy of Bytecode Energy Distribution

We evaluate the accuracy of the bytecode regression model in two ways. First, we determine the accuracy of the regression model at an application level by comparing the amount of application energy calculated using the regression model versus the amount of application energy actually measured during the experiments. Second, we look at the accuracy of the regression model at a path level by determining the multiple correlation coefficient of the calculated paths values. The results of these experiments are reported in Table 2. Note that we cannot perform this experiment at the bytecode level because we do not have a way to establish the bytecodes' ground truth measurements.

To determine the accuracy of the regression model at the application level, we calculate the Accumulated Estimating Error (AEE). This value represents the normalized difference between the amount of energy that the regression model would calculate for the application versus the amount of actually measured energy. Intuitively, the AEE can be thought of as the amount of total energy not accurately accounted for by the regression model and a lower ratio is a stronger result. The AEE is shown in Equation 6. The $\hat{y}_i$ represents the energy calculated for the ith path based on the regression model and y_i is the measured energy of the ith path produced by the Path Adjuster (*i.e.,* with the invocation energy removed.)

$$AEE = \frac{|\Sigma\hat{y}_i - \Sigma y_i|}{\Sigma y_i} \quad (6)$$

To determine the accuracy of the regression model at a path level, we calculate the multiple correlation coefficient (R^2). The R^2 is a well-known statistical measure that shows how well a variable can be predicted based on a linear function of multiple variables. In our case the value to be predicted is the energy of the ith path ($\hat{y}_i$) calculated with the regression model, and the multiple variables are the individual bytecodes solved for during the regression. The value of R^2 is obtained by calculating the solution to Equation 7. As with the AEE in Equation 6, y_i is the adjusted measured value of the ith path, $\hat{y}_i$ is the path's value calculated using the regression model, and $\bar{y}$ is the mean of the measured paths. The R^2 value is measured on the interval $[0, 1]$ with a value close to 1 representing a strong fit of the data to the model and 0 representing a weak fit.

$$R^2 = 1 - \frac{\Sigma(y_i - \hat{y}_i)^2}{\Sigma(y_i - \bar{y})^2} \quad (7)$$

For the AEE, the numbers range from 3.4% to 8.6% with an average of just under 6%. This means that, on average, 6% of the actual measured energy is not accounted for by the model. In general, since linear regression is an approximation, a certain amount of variance in the energy totals is to be expected. In this context, we believe that 6% represents a low enough number that the overall results of the analysis would still be informative and help to guide the developer in making changes to the code. The R^2 values range from 0.9 to 0.99 with an average of 0.936. The high average R^2 value shows that our model is able to fit the measured data very closely. Overall, these numbers show that our regression model exhibits high accuracy with respect to its calculations at both the application and path level.

4.4.3 Accuracy of Outlier Detection

We evaluated the accuracy of the RLR techniques to detect outliers caused by garbage collection and thread switching events. To do this, we seeded a variable number (20, 50, 100, and 200) of artificial events along paths of the subject applications by inserting calls to garbage collection (`System.gc()`) and thread switching instructions (`Thread.start()` and `Thread.join()`). The target thread contained only a single instruction so that its actual execution time would be low and the most prominent cost would be the context-switch overhead. We then applied the RLR techniques and checked to see if the paths containing the seeded events were determined to be outliers. *For each of the applications, the RLR techniques were able to detect all of the seeded high-energy events.* From this result, we conclude that the use of statistical outlier detection to identify paths affected by garbage collection and thread switching overhead is both an effective and practical approach.

4.4.4 Threats to Validity

In this section we discuss the threats to the validity of our empirical evaluation and explain how we addressed these issues.

External Validity: The subjects in this study are real world marketplace apps downloaded from the Google Play Market. They represent different app domains and, in terms of size, are representative of many apps we saw in the marketplace.

Internal Validity: In general, to help ensure internal validity, the accuracy and timing experiments were repeated multiple times and averaged. To ensure that the outlier detection was a result of our technique, we tracked the seeding of the garbage collection and thread switching, so we could determine easily in which paths they occurred. Furthermore, we compared actual and seeded garbage collection events, to verify that they had similar energy characteristics. For threads, we used a minimal sized thread (one instruction) to ensure that the thread's energy was not inflated and therefore made easier to detect.

Construct Validity: For accuracy, we were unable to use the most straightforward comparison, bytecode ground truth, due to the difficulty of measuring the ground truth for such short events. Instead, we used invocation ground truth and well-known statistical tests to show the degree to which our models fit the measured data. Overall, it seems likely that if the calculated values for invocation were close to ground truth and the R^2 values were consistently high across all apps, then the calculated source line level energy would also be accurate. Nonetheless, this is a validity threat that our experimental design could not completely address.

5. RELATED WORK

There are two areas of related work for our approach, energy estimation and power measurement. The first area, energy estimation, assumes that developers do not have access to power measurement hardware and uses software based techniques to predict how much energy an application will consume at runtime. The second group of techniques, power measurement, makes use of power measurement hardware to obtain power samples and then uses software based techniques to attribute the power to implementation structures.

The general process of energy estimation techniques is to build a parameter-based model of energy consumption, capture values for the model's parameters from the application, and then calculate

the values produced by the model. As compared to our approach, the primary difference is that estimation techniques assume that developers do not have access to power measurement devices, the target hardware platforms, or are unable to use these techniques for various reasons. These scenarios can occur frequently in practice and make estimation techniques a complementary approach to ours.

Our prior work developed an approach, *eLens*, for predicting the energy consumption of smartphone applications based on an expected workload [12]. The *eLens* technique is based on per-instruction cost functions, which are provided by a Software Energy Environment Profile (SEEP) and driven by parameters obtained via program analysis. Developing the SEEP is labor intensive and may not always be feasible, as there are thousands of APIs in the Android SDK, many of which require complex energy models. In contrast, *vLens* takes live energy measurements and attributes them to source lines using regression and statistical techniques. The *vLens* technique does not require a SEEP, but does require a Power Measurement Platform, which may be easier to obtain for some developers. Because *vLens* is based on runtime measurements, it must also deal with the influence of garbage collection, thread switching, tail energy, and sampling intervals.

Seo and colleagues [27, 28, 29] have proposed several software energy estimation techniques. They model the energy consumption of each Java bytecode and the network utilization, and capture these two types of information by modifying the JVM. Compared to *vLens*, their estimation requires modifications to the runtime systems, the construction of energy models, and provides information at a component level instead of the source level.

Other approaches estimate software energy using models based on operating system (OS) level features. These features include calls to OS-level APIs [18, 33] or state metrics internal to the operating system [23, 24, 39], such as CPU frequency and network usage. Similar to this work are approaches to estimate the energy consumption in virtual machines [10, 17, 32, 38]. Compared with our approach, this body of work requires significant integration and modification to the runtime systems and is not as portable. Furthermore, the workload of building models can be high.

Another group of approaches for software energy estimation is based on low-level hardware modeling. Works of this type build models based on the assembly instructions [35, 36] or micro-instructions of each assembly instruction [19, 31]. The drawback of these approaches is that they cannot estimate the energy of external devices (*e.g.,*, GPS, WIFI, etc.), which also consume a large portion of energy in modern mobile systems. Moreover, Sinha and colleagues [30] have shown that, at this level, the energy consumption of instructions or micro-instructions are roughly the same. Unlike this body of work, *vLens* can measure the energy consumption of the entire mobile system, including external devices.

The last type of work to estimate software energy is to build simulators. The main drawback of simulators is their speed. Cycle-level simulators [7, 22] can require thousands of instructions to simulate one instruction, which is too slow to use for simulating modern interactive applications. Even functional simulators [21], which are comparatively faster than cycle accurate simulators, run too slowly to be useful for capturing realistic user interactions.

The most closely related techniques to our approach are based on power measurement. The general approach for these techniques is to use a power measurement device, such as the LEAP [25] or Monsoon [2] power meters, that can sample energy measurements at a certain frequency. These measurements are then combined with software based techniques to provide useful information to software developers. Sahin and colleagues [26] map energy consumption to different component level design patterns. Their work explored the correlations between energy consumption and the use of different design patterns. At a higher level, Flinn and colleagues [6, 11] measured the energy consumption of applications and mapped the energy to individual OS processes. This was done, in part, by instrumenting the operating systems.

Sesame [8] measures the energy by reading the battery interface of laptops and uses regression analysis to automatically generate energy consumption information based on system level metrics, such as CPU workload and network utilization measurements. However, Sesame does not provide information at the level of source code lines. Tan and colleagues [34] also use regression analysis to map energy to paths in the control flow graph, but are not able to provide any finer level of granularity with their measurements.

Compared with these measurement approaches, *vLens* is able to provide energy measurements at a much finer level of granularity. *vLens* calculates energy measurements at the source line level, whereas the above mentioned approaches compute their information at the level of entire application, architecture level components, design patters, or methods.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented our approach for calculating the source line level energy consumption of Android applications. Our approach employs a combination of hardware-based energy measurements and efficient path profiling to correlate energy measurements with the application's execution. Then the approach analyzes each of the executed paths to handle high energy events, such as garbage collection and thread switching. The adjusted measurements are then used to perform a regression analysis that maps the energy to the individual source lines.

For the evaluation, we implemented our approach in a prototype tool, *vLens*, and ran it against five real-world marketplace apps. Our evaluation showed that *vLens* was fast, it was able to calculate energy information for each app in under three minutes. The approach was also accurate. It was able to attribute invocation information to within 10% of the ground truth, the statistical models fit the data closely with an R^2 average of 0.93, and it could detect the high energy events with 100% accuracy. Overall, the results are promising and indicate our approach has the potential to help developers understand the energy related behavior of their applications.

In future work we will focus on improving several aspects of our approach. The first aspect is to design new experiments to improve the construct validity of our experiments. Ideally, we would do this via faster power sampling techniques; however, it is unlikely that power samplers will show the same increase in speed as CPUs. Therefore, we will investigate new statistical and experimental methods that will allow us to more precisely quantify source line level accuracy of our approach. Second, we would like to improve techniques for solving the linear systems of equations when there are insufficient data points and attaching "confidence" metrics to our approximations. Third, we would like to explore alternative ways of handling paths that are identified as outliers by the statistical analysis. For example, by identifying ways to attribute high-energy events to source lines in a way that is meaningful for developers. Lastly, we would like to use *vLens* in developer studies to determine its usefulness for making energy consumption improvements.

7. ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under Grant No. CNS-905596 and a Zumberge Research Award from the University of Southern California.

8. REFERENCES

[1] Dex2jar. `http://code.google.com/p/dex2jar/`.

[2] Monsoon. `http://www.msoon.com/LabEquipment/PowerMonitor/`.

[3] The r project for statistical computing. `http://www.r-project.org/`.

[4] Apache. Bcel library. `http://bcel.sourceforge.net/`.

[5] T. Ball and J. Larus. Efficient Path Profiling. In *MICRO 29*, pages 46–57. IEEE Computer Society, 1996.

[6] F. Bellosa. The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems. In *the 9th workshop on ACM SIGOPS European Workshop*, pages 37–42. ACM, 2000.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 83–94. ACM, 2000.

[8] M. Dong and L. Zhong. Sesame: Self-Constructive System Energy Modeling for Battery-Powered Mobile Systems. In *Proc. of MobiSys*, pages 335–348, 2011.

[9] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, Nov. 1992.

[10] K. Farkas, J. Flinn, G. Back, D. Grunwald, and J. Anderson. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. *ACM SIGMETRICS Performance Evaluation Review*, 28(1):252–263, 2000.

[11] J. Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10. IEEE, 1999.

[12] S. Hao, D. Li, W. G. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proc. of 35th International Conference on Software Engineering*, 2013.

[13] P. Huber. Robust statistics. 1981.

[14] Intel. Atom N550 Datasheet. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/atom-n400-vol-1-datasheet-.pdf`.

[15] Intel. X18-M/25-M SATA SSD Datasheet. `http://download.intel.com/design/flash/nand/mainstream/mainstream-sata-ssd-datasheet.pdf`.

[16] X. Jiang, P. Dutta, D. Culler, and I. Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 186–195. IEEE, 2007.

[17] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. Bhattacharya. Virtual Machine Power Metering and Provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 39–50. ACM, 2010.

[18] T. Li and L. John. Run-time Modeling and Estimation of Operating System Power Consumption. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):160–171, 2003.

[19] H. Mehta, R. Owens, and M. Irwin. INSTRUCTION LEVEL POWER PROFILING. In *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, volume 6, pages 3326–3329. IEEE, 1996.

[20] Micron. DDR3 SDRAM SODIMM Datasheet. `http://download.micron.com/pdf/datasheets/modules/ddr3/jsf16c256x64h.pdf`.

[21] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Proc. of MobiCom*, 2012.

[22] T. Mudge, T. Austin, and D. Grunwald. The reference manual for the sim-panalyzer version 2.0 [z]. `http://www.eecs.umich.edu/~panalyzer`.

[23] A. Pathak, Y. Hu, M. Zhang, P. Bahl, and Y. Wang. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *Proc. of EuroSys*, pages 153–168. ACM, 2011.

[24] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof. In *Proc. of EuroSys*, 2012.

[25] P. Peterson, D. Singh, W. Kaiser, and P. Reiher. Investigating energy and security trade-offs in the classroom with the atom leap testbed. In *4th Workshop on Cyber Security Experimentation and Test (CSET)*, pages 11–11. USENIX Association, 2011.

[26] C. Sahin, F. Cayci, I. L. M. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Initial explorations on design pattern energy usage. In *First International Workshop on Green and Sustainable Software (GREENS)*, pages 55–61, 2012.

[27] C. Seo, S. Malek, and N. Medvidovic. An Energy Consumption Framework for Distributed Java-Based Systems. In *Proc. of 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 421–424. ACM, 2007.

[28] C. Seo, S. Malek, and N. Medvidovic. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. In *Proc. of 11th International Symposium on Component-Based Software Engineering*, pages 97–113. Springer, 2008.

[29] C. Seo, S. Malek, and N. Medvidovic. Estimating the Energy Consumption in Pervasive Java-Based Systems. In *Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, pages 243–247. IEEE, 2008.

[30] A. Sinha and A. Chandrakasan. Jouletrack-A Web Based Tool for Software Energy Profiling. In *Design Automation Conference, 2001. Proceedings*, pages 220–225. IEEE, 2001.

[31] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proc. of PATMOS*. Citeseer, 2001.

[32] J. Stoess, C. Lang, and F. Bellosa. Energy Management for Hypervisor-Based Virtual Machines. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, page 1. USENIX Association, 2007.

[33] T. Tan, A. Raghunathan, and N. Jha. Energy Macromodeling of Embedded Operating Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(1):231–254, 2005.

[34] T. Tan, A. Raghunathan, G. Lakshminarayana, and N. Jha. High-level Software Energy Macro-modeling. In *Proc. of Design Automation Conference (DAC)*, pages 605–610. IEEE, 2001.

[35] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power

Minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[36] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee. Instruction Level Power Analysis and Optimization of Software. *The Journal of VLSI Signal Processing*, 13(2):223–238, 1996.

[37] J. Turkey and A. Beaton. The fitting of power series, meaning polynomials, illustrated on band-spectroscopic data. *Technometrics*, 16:189–192, 1974.

[38] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. Irwin. Energy Behavior of Java Applications from the Memory Perspective. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 23–23. USENIX Association, 2001.

[39] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc. of IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 105–114. ACM, 2010.