Estimating Mobile Application Energy Consumption using Program Analysis

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

***Abstract*—Optimizing the energy efficiency of mobile applications can greatly increase user satisfaction. However, developers lack viable techniques for estimating the energy consumption of their applications. This paper proposes a new approach that is both lightweight in terms of its developer requirements and provides fine-grained estimates of energy consumption at the code level. It achieves this using a novel combination of program analysis and per-instruction energy modeling. In evaluation, our approach is able to estimate energy consumption to within 10% of the ground truth for a set of mobile applications from the Google Play store. Additionally, it provides useful and meaningful feedback to developers that helps them to understand application energy consumption behavior.**

***Index Terms*—Mobile app, fine-grained energy estimation, program analysis.**

I. Introduction

Smartphones and tablets allow people to carry around more computational power in their hands than most had on their desktops just a few years ago. However, the usability of these devices is strongly defined by the energy consumption of mobile applications, and user reviews of applications reveal many customer complaints related to energy usage.

Research in estimating the energy usage of mobile devices has explored a wide variety of techniques, ranging from specialized hardware, cycle-accurate simulators and operating system level instrumentation, to carefully calibrated software-based energy profilers that provide coarse-grained energy estimates. From the perspective of a developer wishing to optimize energy consumption of an application, each of these approaches has one or more shortcomings: specialized hardware can be expensive, cycle-accurate simulators and operating system level instrumentation can slow down a mobile app beyond the point of usability, and coarse-grained energy estimates may not be able to pinpoint hotspots within an app.

To address these shortcomings, we explore a novel approach, called *eLens*, that combines two ideas that have not previously been explored together: *program analysis* to determine paths traversed and track energy-related information during an execution, and *per-instruction energy modeling* that enables *eLens* to obtain fine-grained estimates of application energy. *eLens* does not require the developer to possess specialized hardware or to instrument the operating system, does not impact the usability of applications, and can measure energy-usage at method, path or line-of-source granularity.

These two ideas work in concert as follows. When a developer wishes to obtain an energy estimate for specific use cases of a mobile app, *eLens* uses instrumentation to identify the corresponding paths of an application that will be executed and record runtime information that is needed by the energy models (Section III). To compute the energy estimate, *eLens* analyzes the recorded paths and runtime information to extract the energy relevant information, and uses this information to drive the energy models and estimate the energy consumption of each bytecode or API call for the hardware components of the system (e.g., CPU, memory, network and GPS). The energy models are provided by a software environment energy profile (SEEP), whose design and development enables the per instruction energy modeling (Section IV). *eLens* energy consumption estimates can be computed at different levels of granularity, application, method, path, and line of source code and integrated into a development environment, such as Eclipse, so that developers can visualize the energy usage of their application during development.

eLens has several desirable properties that distinguish it from prior work. By design, it is *lightweight*; *eLens* does not require modifications to the mobile operating system or require expensive power monitoring hardware. Moreover, *eLens* provides *fine-grained* visibility into the energy consumption of an application at multiple levels of granularity down to an individual line of source code. Using experiments on popular mobile applications obtained from the Android marketplace, we demonstrate two other important properties (Section V). *eLens* is *accurate*; it is able to estimate the power consumption of real marketplace applications to within 10% of ground-truth measurements. Competing methods that are path-insensitive, or use coarse-grained energy models can be an order of magnitude more inaccurate. Finally, it is *fast*, allowing developers to easily analyze the energy behavior of multiple combinations of hardware and operating systems.

II. Related Work

Power modeling is a broad area of research that encompasses several sub-fields, including architecture, operating systems, and software engineering. Given space limitations, we present representative pieces of work from each area to place *eLens* in the context of prior work.

Closely related to *eLens* is the body of prior work on architectural power modeling, which has attempted to model or profile the power consumption of individual instructions. Tiwari and colleagues [28][29] model the energy of an instruction using a base energy as well as a transition energy for each pair of instructions. Steinke [24] discusses a more detailed power model that takes architectural features, such as pipeline stalls, into account. Sinha and colleagues [23] show that ARM instructions all consume comparable energy. Finally, Mehta and colleagues [13] profile energy usage at the level of architectural functional units. In contrast to this body of work, *eLens* estimates bytecode energy costs; energy of bytecodes shows considerably higher variation and may be less affected by architectural effects at the instruction-level.

Cycle-accurate simulators have also been developed to estimate software energy consumption of software (such as Sim-Panalyzer [15] and Wattch [5]). These approaches can simulate the actions of a processor at an architecture-level and estimate energy consumption in each cycle. Compared to *eLens*, these methods can be highly inefficient (e.g. Sim-Panalyzer needs 4300 instructions to simulate the execution of a single instruction) impeding their usability for complex mobile applications that involve user interaction.

Novel hardware designs have been proposed to estimate energy consumption. The LEAP platform [18], which we use in this work, provides fine-grained measurements of energy. Others have designed an FPGA-based embedded device to perform component-wise energy profiling and empirically measured the energy impact of using different software design patterns [19]. By contrast, *eLens* requires no specialized hardware to obtain fine-grained estimates of app energy usage.

Other works focus on the energy consumption of the operating system at routine or system-call level ([6], [12], [26], [31]). They all build power models at the system or routine call level, which describe the power consumed as a function of some feature of the system or routine call (e.g., the CPU utilization or the input parameters). These power models are then used to estimate system-level energy consumption. *eLens* is inspired by this work, but builds models at the instruction granularity, and so is able to estimate power down to the granularity of a line of source code.

A complementary approach [16], [17] explicitly models the state transitions between hardware power states of smartphone hardware components (CPU, WiFi, GPS etc.). This approach then estimates the power states of each component during a system call, based on the call input parameters. Using this and measured values of hardware power states, it is possible to compute the approximate energy consumption of applications and functions that invoke system calls. However, unlike *eLens*, this requires manual instrumentation of the application framework and may not work for applications with energy-intensive application level code.

Also complementary is recent work [14] that allows developers to estimate overall application energy usage using an emulator. In contrast, *eLens* can be integrated into an IDE and provides much finer-grained energy estimates, making it more seamless for the developer to optimize applications for energy.

Beyond instruction-level and call-level energy modeling, some work has also considered path energy profiling. Tan and colleagues [27] model path energy costs using the Ball-Larus profiling technique [3]. In comparison, *eLens* directly estimates bytecode costs and environment invocations, so it can be used to compute energy utilization at different granularities.

Tangential to *eLens* is a body of work that has attempted to estimate the energy consumption of Java on different virtual machines ([8], [30], [25], [11]). *eLens* provides fine-grained estimates of energy usage within an application.

Like *eLens*, Seo and colleagues [20], [21], [22] built an instruction-level model of Java bytecodes and linear model of interfaces of the JVM and operating system-calls. Unlike *eLens*, however, they require modifications to the JVM to estimate bytecode costs online. Moreover, they do not perform any path-sensitive analysis and so are able to provide energy consumption estimates only at the system level.

Complementary methods for estimating the energy usage of applications have relied on operating system level instrumentation [9], [4], together with hardware support for energy measurement. *eLens* relies on power profiles and does not require modifications to the operating system.

Finally, in a previously published workshop paper [10], we described preliminary work that used execution traces to estimate CPU energy consumption. This paper extends that work by improving the underlying analyses, adding a more sophisticated CPU model that accounts for frequency scaling, and expanding the technique to include other hardware components, such as RAM, WiFi, and GPS, via the SEEP. Furthermore, *eLens* is able to handle real marketplace applications, whereas the previous work could only estimate energy for instructions that were not invocations.

III. Our Approach for Energy Estimation

eLens analyzes the implementation of a mobile application and provides code-level estimates of the energy that it will consume at runtime. The results of this analysis are summarized at the granularity of the whole program, path, method, and source line to help the developer make informed implementation decisions for reducing energy consumption. The inputs to the approach (Figure 1) are: (1) the software artifact; (2) the workload, which describes the way the software will be used at runtime; and (3) system profiles, which use per-instruction energy models to specify the power characteristics of the platforms for which the developer is targeting the implementation (Section IV). Within *eLens*, there are three components: the Workload Generator (Section III-A) translates the workload into sets of paths through the software artifact; the Analyzer (Section III-B) uses the paths and system profiles to compute an energy estimate; and, the Source Code Annotator (Section III-C) combines the paths and energy estimate to create an annotated version of the source code that is provided to the developer. The output of *eLens* is a visualization that shows the estimated energy consumption of the software at the path, method, source line, and whole program granularity.

A. Generating the Workload

The Workload Generator is responsible for converting the user-level actions, for which the developer wants an estimation, to the path information used by the Analyzer and Source Code Annotator. Trivially, this information could be provided by assuming that every path in the artifact is executed *n*

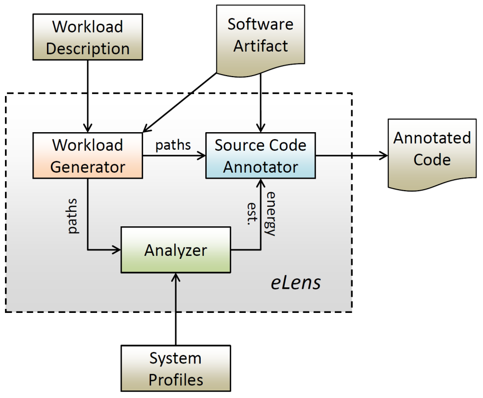


Fig. 1: Overview of *eLens*

number of times. However, this would not be an accurate reflection of how the application would execute at runtime, and since energy consumption is not uniform over different paths, the resulting estimate would not be as helpful. For example, in a video viewing application, the paths traversed to watch a video will consume more power than the paths traversed to start or exit the application.

The inputs to the Workload Generator are the workload description, W, and the implementation of the application, S. The *workload description* is a specification of the behavior of the application for which the developer wants to estimate energy consumption and can be represented as a sequence of use cases $\langle u_1, u_2, ..., u_n \rangle$. The Workload Generator instruments S to create a version, S', that will record the paths traversed during an execution. Next, the Workload Generator runs each $u_i \in W$ on S'. The paths traversed for u_i are denoted as P_i. The set of all P_i, $\mathscr{P}$, is the output of the Workload Generator.

The workload description can be specified informally, where the developer simply interacts with the instrumented application, or formally, where the sequence of actions is explicitly listed and can be executed by automated Android testing tools, such as MonkeyRunner [1] and Robotium [2]. We only require that the specification mechanism must be able to execute the instrumented version of the application, so the paths traversed by the application are recorded. There is no adequacy criterion for the workload description except that it represents the set of actions that are of interest to the developer. For example, an informal workload description of a video player may specify that the user perform the following actions: (a) start the application, (b) search for a video using the term "*eLens* at ICSE", (c) play the first video found, (d) replay the video 100 times, and (e) exit the application.

The instrumentation inserted by the Workload Generator records the path traversed through each method of the application. This recording is based on an efficient path profiling technique proposed by Ball and Larus [3]. The Ball-Larus approach assigns weights to edges of a method's control-flow graph (CFG) such that the sum of the edge weights along each unique path through the CFG results in a unique path ID; a single instrumentation variable per method then suffices to record the traversed path for one method invocation. Therefore, each P_i is comprised of a sequence of sub-path tuples, each denoted by $\langle m, id \rangle$, where m is the method and id is the ID of the path traversed. Our implementation extends the Ball-Larus approach to handle nested method calls, concurrency, and exceptions, as described in Section V-A.

To illustrate the output of Workload Generator, consider a software artifact with three methods a, b and c. For this artifact, a possible $\mathscr{P}$ is $\{\langle\langle a,1\rangle,\langle b,1\rangle,\langle a,2\rangle\rangle,\langle\langle a,2\rangle,\langle c,3\rangle\rangle\}$, which contains two paths. The first corresponds to a use case in which path 1 of method a is executed followed by path 1 of method b and then path 2 of method a. The second corresponds to a use case in which path 2 of method a is executed followed by path 3 of method c.

Algorithm 1 Estimate Energy Consumption

```
Input: $H$: set of hardware components, $l$: source line,
       $m$: method, and $P$: path
Output: Energy estimate in Joules
1: $cost \leftarrow 0$
2: $\langle \Theta, M, L \rangle \leftarrow regenerate(P)$
3: $D \leftarrow propagateDT(\Theta)$
4: for all $h \in H$ do
5:     for all $i \in \Theta$ do
6:         $f_h \leftarrow powerstate(i,h)$
7:         if $L(i) = l \wedge M(i) = m$ then
8:             $cost(h) += C(i,h,f_h,D(i))$
9: return $\sum_{h \in H} cost(h)$
```

B. Estimating Energy Consumption

The Analyzer computes energy estimates using the path information provided by the Workload Generator and the energy cost functions from a software environment energy profile (SEEP) (Algorithm 1). As input, the Analyzer takes a sequence of sub-paths $P \in \mathscr{P}$, method m, source line number l, and set of hardware components H to be accounted for in the estimation. For each instruction[1] specified by l and m, the Analyzer calculates the energy cost using the cost functions defined in the SEEP. The output of the analysis is then the energy estimate E, in Joules, of the path, method, or source line. An energy estimate for the entire software artifact can be calculated by summing the calls for each $P_i \in \mathscr{P}$.

The SEEP defines energy cost functions $C(\cdot)$ for all instructions. Broadly speaking, there are two dimensions along which energy costs may vary for instructions, power state and path-dependent information. Most components, such as the CPU, network and GPS, have multiple *power states*: modern smartphone CPUs can operate at different frequencies, which consume different amounts of energy; the GPS may be on or off; and the network may be idle, transmitting/receiving, or have selectively enabled multiple antennas. Thus, $C(\cdot)$ can depend upon the power state of the corresponding component when a bytecode or API component is executed. Furthermore, some API invocations' energy consumption is based on path-dependent information. For example, sending data over the network incurs energy costs (roughly) proportional to the size of the data transfer. During the process of workload generation,

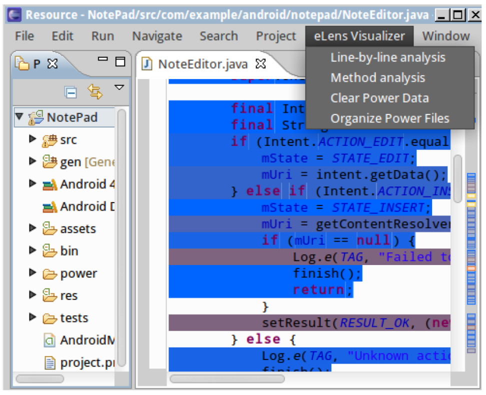
[1]We use *instruction* to denote both bytecodes and system API calls.

eLens tracks the power states of hardware components, and also certain types of path related data. This information is included as one or more arguments to the cost functions defined in the SEEP. Details and examples of path-dependent information are provided below and in Section IV.

The first operation performed by the Analyzer is to regenerate the instruction sequence represented by P (line 2 of Algorithm 1). As defined by the Ball-Larus algorithm, given a subpath $\langle m, id \rangle$, it is possible to regenerate the sequence of instructions that define each subpath. The *regenerate* combines each subpath in P to define the complete path (entry to exit) taken during the execution. For nested method calls, where method A calls method B, *regenerate* calculates the sequence of instructions $a_1, a_2, \ldots, a_n$ traversed in A and $b_1, b_2, \ldots, b_m$ traversed in B. If a_k is an invocation to B, then the final sequence would be $a_1, a_2, \ldots, a_k, b_1, b_2, \ldots, b_m, a_{k+1}, \ldots, a_n$. Identifying which subpaths in P were called by other subpaths is straightforward, since P is a sequence and each $\langle m, id \rangle$ is appended to P after m exits. The final output of *regenerate* is the tuple $\langle \Theta, M, L \rangle$, where is Θ is the complete path, M maps each instruction in Θ to its containing method, and L maps each instruction in Θ to its source code line number.

The energy cost of certain instructions is based on path-dependent information. For example, the cost of a network send instruction depends upon the amount of data sent and the cost of opening an input stream will depend on the stream type. Line 3 of the algorithm propagates argument and type data along Θ that can be used to identify this type of information and initializes a function D that relates this information to each instruction. The *propagateDT* function implements this functionality by simulating data-flow along the path Θ. The instrumentation introduced by the Workload Generator records information, such as the size of data operated on by an instruction or the class implementing an API call at specific points of the execution. *propagateDT* then simulates the stack contents along Θ to track the types and relevant data attributes to the point where they are needed by the energy cost functions. Stack simulation works well in this context because it is operating on only one complete path (Θ) and only a subset of all instructions on the path need to be simulated. In cases where values are manipulated by an uninstrumented function (e.g., a library call), an average value is used for the energy cost functions. More details on the specific types of information tracked is discussed in Section IV.

Many path-dependent APIs require specialization of the general approach described above. Due to space constraints, we provide two representative examples of the analysis employed by the *propagateDT* function. (1) Precisely identifying the class that extends `InputStream` is difficult because in Android applications these often originate from a factory class. To gather this information the Workload Generator inserts a probe into S' after calls to specific factory methods to record the implementing class type. This information is then used to annotate the data item in the stack simulation, so that when the `InputStream` API is called, the implementing class on the stack can be identified and the appropriate cost functions used. (2) Allocation instructions set aside memory space for an array. For example, network buffers may allocate a byte array and define its size using a hard-coded constant of 1024. The *propagateDT* function tracks loads of constants onto the stack so that when they are popped and used as arguments, the value of the constant can be identified. In the example above, the Analyzer would be able to determine that 1024 is the value of the constant on the stack used to supply the argument to the allocation statement.



```
        final Int
        final Str
        if (Intent.ACTION_EDIT.equal
            mState = STATE_EDIT;
            mUri = intent.getData();
        } else if (Intent.ACTION_IN
            mState = STATE_INSERT;
            mUri = getContentResolve
            if (mUri == null) {
                Log.e(TAG, "Failed t
                finish();
                return;
            }
            setResult(RESULT_OK, (ne
        } else {
            Log.e(TAG, "Unknown acti
```

Fig. 2: Source line visualization provided by *eLens*

As described above, many components have multiple power states. For example, modern CPUs conserve power by changing the CPU frequency in response to high or low utilization. An instruction's power consumption can therefore depend on the frequency of the CPU when it is executed. The function *powerstate*, used in line 6, maps each instruction $i \in \Theta$ to the power state f_h of component h (where h may be the CPU, WiFi, or other component with multiple power states) when i was executed. *eLens* computes f_h by tracking, during workload generation, when component power state changes occur. The cost functions for each instruction take f_h and h as arguments. For example, assuming a CPU has two frequency levels, high and low, the CPU cost function for an `ldc` instruction would return two different energy cost values e_{high} or e_{low}, depending on whether f_{CPU} reported the frequency as high or low for that instance of the instruction.

Each instruction that satisfies the method and line number constraints is added to the total energy cost (lines 7–8). The Analyzer calculates an instruction's energy cost for each component of the platform as a function of its type, path-dependent data, and component power state. The Analyzer can be configured to explore different sets of hardware components via the input H.

C. Energy Annotations

The Source Code Annotator converts the path information and energy estimation numbers into a graphical representation that allows developers to visualize the energy consumption of their software. The representations connect the estimated energy numbers to the implementation structure of the software artifact. The ability to visualize energy usage at a source code line level is a unique feature of *eLens*. Feedback at this level allows developers to iteratively refine implementation details, such as statements and loops, to improve the overall energy consumption of the software. We have implemented the visualization as an Eclipse plugin, which can provide visual representation of the energy consumption at four different granularity levels: whole software, per-method, per line of source code, and for each path. Note that, for each granularity level, the energy consumption can be shown for some or all use cases in the workload. The mechanism for two of these representations is discussed below; the mechanisms for path and whole-software annotations follow from these.

Per Line of Source Code: For a given source file, the annotator ranks each source line according to its energy cost over all $P \in \mathscr{P}$. Then the rankings are mapped to a color spectrum, such as blue to red, and each line of source code is colored based on its position on the spectrum. This results in a SeeSoft like visualization [7] of the power consumption of the software. Figure 2 shows a screenshot of a Java source file with the energy based colorings.

Method Representation: For a given source file, the Source Code Annotator generates the call graph (CG) of the software artifact. The methods in the CG are ranked according to their energy consumption over all $P \in \mathscr{P}$ and then assigned a color in a spectrum based on their relative use of energy. A method's assigned color and energy value are then used to annotate the corresponding node in the CG.

IV. Software Environment Energy Profile

The Software Environment Energy Profile (SEEP) provides per-instruction energy cost functions for each component of the target platform. The use of the SEEP allows *eLens* to analyze energy consumption on multiple platforms by simply providing different SEEPs as input to the Analyzer. We anticipate that a SEEP will be developed and distributed by a platform's manufacturers as part of the platform's software development kit. This method of distribution makes it unnecessary for developers to use complicated or expensive energy monitoring equipment. Currently, it is not common practice for manufacturers to provide SEEPs, so we discuss below the steps required to develop a SEEP.

For each distinct hardware component, the SEEP contains a function that estimates the energy cost of each instruction at each distinct power state of the hardware component. Thus, $C_{CPU}(i,f)$ denotes the CPU energy cost of instruction i at frequency f and $C_{WiFi}(i)$ the WiFi cost[2].

To estimate these cost functions, we used the LEAP power measurement device [18]. LEAP reports energy-consumption measurements at a fine-granularity, for each hardware component in the system; its analog-to-digital converter (DAQ) samples each component's current draw at 10 KHz. It contains an ATOM processor that runs Android version 3.2, so we can measure the energy consumption of Android applications. In addition, the LEAP provides Android applications with the ability to trigger a pulse that can be used to correlate application activity with DAQ readings. This capability allows us to time specific sections of profiling code by generating pulses at the beginning and end of sections of code whose energy consumption we wish to measure. The DAQ readings are stored in a log and post-processed in order to estimate the total energy consumed between a pair of pulses. These measurements are performed by hardware external to the components.

[2]The latest WiFi standard, 802.11n, supports multiple power states. We have left cost functions for 802.11n as future work.

The energy cost functions for instructions can be broadly grouped into two categories: those with a path-dependent energy cost and those with a path-independent energy cost. The cost functions for the latter can be approximated for each hardware component by knowing only information local to the instructions. The path-dependent instructions require additional information that can only be identified by incorporating information from prior instructions in the executed path. The profiling for both types is explained in more detail below.

A. Path-Independent Cost Functions

Cost functions for instructions with path-independent costs can be calculated for each hardware component using the type of instruction and power state (e.g., CPU frequency) of the hardware component at which the instruction was executed. We identified these instructions by analyzing the implementation of the Dalvik Virtual Machine and confirmed our analysis through empirical measurements. To profile individual instructions, we created a set of test scripts, each of which profiles an individual instruction by placing that instruction in a loop that executes 20 million times. This ensures a measurement of a sufficiently long duration that exceeds the DAQ sampling interval and reduces variance. Each loop is then run ten times on a quiescent system to minimize the impact of cold starts, cache effects or background processes (such as garbage collection). We subtract the cost of the loop setup and LEAP pulse instructions from the measured cost.

Some instructions require the results of executing other instructions. For example, an `add` instruction requires that two operands be pushed on to the stack. To handle this, a dependency is identified between the test scripts. For instance, if the `ldc` instruction is used to push operands on the stack, then the test script for the `add` instruction includes these bytecodes. When profiling the `add` instruction, we subtract the cost of the two `ldc` instructions. We repeated this process for all of the Dalvik bytecodes we identified as having a local cost. Details for different categories of bytecodes are provided below. This list is not intended to be exhaustive, but to illustrate how different categories were handled:

Invocations and returns from application function: When the target of an invocation is an application function, the target method will already be instrumented by the Workload Generator, so only the overhead of the invocation and return needs to be profiled. In the profiling scripts, it was not possible to isolate an invoke from a return; both are required to execute in pairs. Therefore, we measured all combinations of invoke and return types and then calculated the cost of each instruction. Values for these instructions varied according to the type of invoke (e.g., virtual or static) and the type of value returned (e.g., void, integer, or float).

Invocations to fixed-cost APIs: Many invocations to system APIs performed operations, such as setting a property, that could be described with a fixed cost. These were profiled differently than regular instructions because of their relatively high execution time, both in terms of the functionality they implemented and argument initialization. We profiled those APIs that had a fixed cost over all hardware components and were called from within one of the applications used in our evaluation. This set included approximately 1,500 unique APIs. For these, we instrumented our application to obtain energy consumption incurred by each API, and ran our applications several times to obtain a sufficient number of energy consumption samples for each API call. The average of these samples for each API was used as its fixed cost.

Other instructions: The energy costs for loads and stores varied according to the basic type on which they operated (e.g., integer or float). For arithmetic and logic instructions, the energy cost varied based on the operation performed and the basic type on which the operations were performed. Stack management instructions had a fixed cost. Finally, jumps and branches incurred a fixed cost regardless of the type of condition they check.

B. Path-Dependent Instruction Costs

The energy cost for path-dependent instructions is based not only on hardware power state and instruction type, but also on information that is provided by other instructions in the path, such as the size of the arguments to a network send instruction. In general, we found four categories of path dependent instructions: allocation instructions, invocations of system APIs whose cost depends on the argument data, invocations of system APIs whose cost depends on the implementing class, and invocations whose cost depends on external data. These types are discussed in more depth below.

Allocation instructions, such as `anewarray`, cause memory to be set aside for an array of basic types. Our measurements indicate that the cost of allocation statements is linearly proportional to the amount of memory allocated. Therefore, the cost of an allocation instruction is a linear function with respect to the size of the array and the size of the basic type allocated. The size of a basic type (e.g., char, int and object reference) is known ahead of time and the runtime instrumentation can record an estimate of the array dimension at runtime, which is then propagated along the path by the *propagateDT* function described in Section III. For allocation of objects, the `new` command has a simpler fixed cost function, as it only initializes a pointer, which is followed by an invocation of the object's constructor, which is handled as an invoke to function.

The energy cost of data-dependent invocations is based on the size of the arguments to the invocation. This type of invocation is often used to access hardware components, such as the network, or perform data manipulations, such as data sorting. For modeling invocations to hardware components, our profiling was informed by research work that investigated and modeled power consumption of various Android hardware components. We identified salient features of these models whose values could be provided via program analysis, and used empirical evaluation of the LEAP node to determine values for hardware-specific constants. Hardware component models were built for the CPU, RAM, WiFi, and GPS. It was not possible to model additional components because they required hardware modification to the LEAP to wire them into the DAQ, a necessary step to verify the accuracy of the models. As we show in Section V, we were able to achieve high accuracy and believe that the process is straightforward to extend to other hardware components as the ability to measure them within the LEAP framework becomes available. To provide the argument data size to the energy cost model, the *propagateDT* function propagates the size of data structures to data-dependent invocations.

The cost incurred by some invocations relies on data or conditions external to the application that cannot be identified using analysis of the path. For example, for an invocation that retrieves a web page or queries a database, there are two associated energy costs: the invocation that makes the request for data and the processing of the response. In our experimentation, we found that the former could be modeled accurately as a function of the response time of the external source of data and the latter as a function of the size of the response. To build the energy cost functions for these invocations, we instrumented all invocations that made external data requests to record the name of the data requested (e.g., URL, database query, or filename), response time, and response size. We used this information to build a map from the data name to its response attributes. Then, when computing the cost of an invocation that depends on external data, we used the map to look up the name of the requested data and provided the response time as an argument to the invocation's energy cost function. When a subsequent invocation occurs that processes or iterates over the returned data, the size of the response is supplied as an argument to this invocation's cost function. As with the previous types of invocations, the response attributes were propagated to the invocation using the *propagateDT* function. We used this methodology to estimate the cost of database calls to the Android SQLite database and invocations to the `HttpMessage` interface.

Invocations of some methods can vary significantly based on the class that provides the method implementation. For example, the cost of accessing the member functions of the abstract class `InputStream` will depend on whether its implementation is provided by a network-based class, such as `ChunkedInputStream`, or a file-based class, such as `FileInputStream`. To handle this type of invocation, we used manual analysis and empirical measurements to identify the methods whose energy varied due to differences in their implementing class. We analyzed each such method and its implementation to determine whether a model could be built based on simply profiling the method or whether a more complex model, based on argument size or external data, needed to be constructed. The energy cost function for these methods could therefore depend on data tracked by the `propagateDT` function, argument data and implementing class information.

V. Evaluation

In this section, we empirically evaluate *eLens* by measuring the *accuracy* of its energy estimates, illustrating the *usefulness* of our approach by evaluating whether time profiling would have been an effective substitute for energy profiling, and demonstrating the *usability* of *eLens* in an interactive development setting by measuring its run time. We conclude with a case study that demonstrates how *eLens* can be used to understand how applications consume energy.

A. Methodology

Our evaluation is based on an implementation of *eLens* which is able to estimate the energy usage of *unmodified* Android applications from the Google Play store. As input, *eLens* takes the implementation of an application in Dalvik bytecode and uses the `dex2jar` tool[3] to convert it into Java bytecode. The Java bytecode version is then provided as input to the Workload Generator and Source Code Annotator. The output of *eLens* is a visualization and reports on the estimated energy consumption of the application.

The Workload Generator uses the `BCEL` instrumentation library[4] to add path profiling instrumentation (as discussed in Section III) and to collect the data and type information as needed for the SEEP. In our implementation, we identify and discard paths that result in exceptions, since their catch blocks may cause control-flow to jump outside of the method, a behavior for which the Ball-Larus algorithm is undefined. Overall, this only resulted in less than 0.01% of paths being discarded. Concurrency is handled by using the thread's ID to identify the counter that must be updated to track the path ID. After instrumenting the application, it is compiled from Java bytecode back to Dalvik bytecode using the standard Android `dx` tool and deployed to the LEAP platform. The use cases were run manually by interacting with the application while it was deployed on the LEAP platform.

The algorithms for the Analyzer and Source Code Annotator were implemented as discussed in Sections III-B and III-C. The Analyzer uses the SEEP that we built as specified in Section IV. The Source Code Annotator was built as an Eclipse plugin and can display energy consumption at the level of granularity of the whole application, method, path, or source line. Screenshots for the plugin are shown in Section III-C.

Subject Applications: Table I shows the set of subject applications that we used in our empirical evaluation. For each application, the table shows the number of classes defined in the implementation (C), the total number of methods across all of those classes (M), number of total bytecodes (BC), and a brief description of the functionality of the application. We used total bytecode count instead of source lines because the apps were downloaded from the Google Play market and source code was not provided as part of the distribution.

We selected the applications based on three criteria: (1) diversity of provided functionality, (2) ability to convert the Dalvik bytecode to Java bytecode, and (3) ability of the application to run on the LEAP platform. The last two criteria curtailed the number of applications available for us to experiment with. `dex2jar` is not yet fully mature and is sometimes unable to completely translate Dalvik to Java bytecode. Furthermore, many applications use native libraries that are not available for the LEAP's x86 processor.

B. Accuracy of eLens

We first compare the accuracy of the estimates produced by *eLens* against the ground truth (GT) measured by the LEAP platform. To do this, we provide each application as input to *eLens*. This generates an instrumented version that is deployed to the LEAP platform, where we interact with each application to exercise its most prominent features. During the execution, the LEAP platform measures power consumption across all of the hardware components. After the execution, the Analyzer computes energy estimates which are summarized and reported by the Source Code Annotator. We compared the measured GT against the *eLens* estimates at both the method and whole software level. As explained below, it was not possible to calculate GT at the line of source code level.

We also compare *eLens* to two other plausible strategies for approximating the energy consumption of mobile applications. The *average-bytecode* strategy does not, unlike *eLens*, use per-instruction energy cost functions, but assumes that each instruction has a uniform cost that can be calculated by averaging the cost of all instructions over all runs of the application. The *no-path-sensitivity* strategy does not, unlike *eLens*, account for the specific paths traversed in a method but estimates energy based on the number of times a method is called multiplied by the cost of all of the method's bytecodes (using per-instruction energy costs). These strategies represent the results that could be achieved by a developer with a method profiler (e.g., gprof) and power measurement device, but each lacks one important capability of *eLens* (per-instruction energy modeling and program analysis, respectively).

The GT is calculated by taking hardware-level measurements on the LEAP platform while the application is running. However, there are three challenges that preclude a straightforward measurement and require a more complex process for establishing GT. First, the applications can pause while waiting for user input or a data response. Although the application is not executing, the device will still consume energy that should not be counted towards an application's GT. Although this idle time is imperceptible to humans, our measurements show that it dominates the total execution time of an application and must be excluded in order to not have a GT dominated by times when the application is idle. Second, even when measuring the application energy during periods when it is not idle, the LEAP (or any power monitor) cannot distinguish if the measured energy was expended by the application or

[3]http://code.google.com/p/dex2jar/

[4]http://commons.apache.org/bcel/

TABLE I: Subject applications

	Application Information				Estimation Error (%)				Time vs. Energy		Timing (s)	
App	C	M	BC	Description	CPU	RAM	WiFi	GPS	r	cos	T_{Inst}	T_{Est}
BBC Reader	590	4923	293910	RSS reader for BBC news	-6.2	5.9	-6.8	-	0	0.21	344	16
Bubble Blaster II	932	6060	398437	Game to blast bubbles	-11.5	3.5	-11.6	-	0	0.01	450	17
Classic Alchemy	751	4434	467099	Game to combine chemical elements	-7.9	-6.9	-4.4	-	0	0.13	886	17
Location	428	3179	232898	Provide location with PL2303 dirver	-7.8	-8.4	-	8.1	0	0.17	274	10
Skyfire	684	3976	274196	Web-browser	-7.9	0.9	-8.4	-	0	0.69	258	8
Textgram	632	5315	244940	Text editor	5.2	4.6	4.6	-	0	0.05	269	6

background processes. Third, the LEAP samples at 10 KHz, so in theory it can only capture energy usage of methods whose execution time exceeds 0.1ms. In practice, we found that reliable energy estimates can only be obtained for functions which run for at least 10 ms. This meant that GT for many methods could not be measured and it was not possible to measure energy consumed by a specific source line of code.

All three of these were addressed by our experiment's methodology. To address idle time we identified and timestamped APIs where the applications could block and idle. Unless other threads in the application were executing during the blocked time, we counted this as idle time and subtracted it from the GT. To ensure that measured energy was accurately attributed, we performed the GT measurements on a quiescent system. As mentioned in Section IV, our technique does not account for garbage collection or process switching, so we identified points during the execution when these occurred and excluded the energy consumed along the affected subpaths from both the energy calculation and the GT total. This represented only 0.05% of the total paths traversed in our experiments. Lastly, to account for sampling frequency, we only conducted accuracy experiments at the whole program and method level. Only methods that ran for more than 10ms at a time were included in the evaluation.

Note that this does not mean *eLens* cannot be used for source line estimates, only that the LEAP platform could not provide GT to evaluate the accuracy of these estimates. *eLens* excludes waiting time energy because it counts only instructions executed by the application, and is able to estimate energy usage of arbitrarily small functions because it uses profiled costs of bytecode energy usage. For the same reason, *eLens* can isolate the energy usage of application code. These are significant advantages of our approach.

Figure 3(a) shows the accuracy at the level of granularity of the whole application. Our subject applications are shown along the X-axis. Figure 3(b) shows accuracy at the method level for those methods whose running time exceeded 10ms. Note that for all methods, there were only six that executed long enough to get an accurate GT measurement. For each application or method, the bars show on a logarithmic scale the average estimation error (compared against GT) of ten runs reported by *eLens* and the two reference techniques, average bytecode and no path sensitivity. Each bar also shows one standard deviation above and below average estimation error.

The results show that *eLens* is able to calculate energy estimates with high accuracy at both the whole program and method level. For the subject applications, *eLens*' estimation error at the whole program level was below 10% across all applications with an overall average of 8.8% (std. deviation of 3%), and from 7.2% to 10%, with an overall average of 7.1% at the method level (std. deviation of 3.6%). Furthermore, *eLens* is able to accurately break down application energy usage by hardware component (Table I); its energy estimation errors for all hardware components are within 12%. Note that Location was the only app that used GPS. This is highly encouraging, and suggests that *eLens* can be a viable approach for exploring the energy usage of mobile applications.

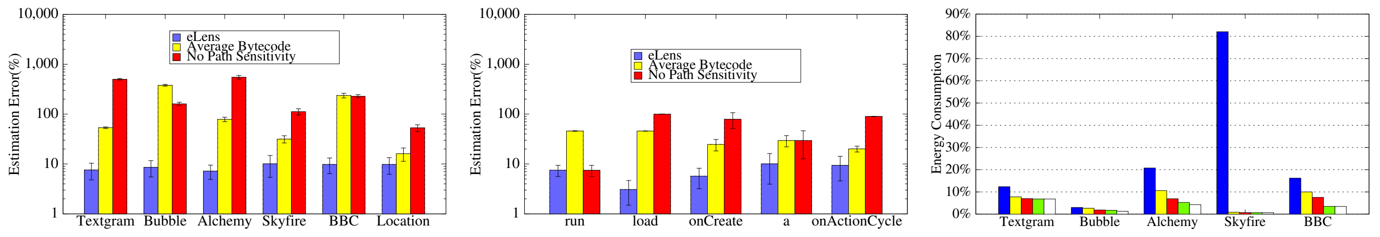
In comparison, the two other plausible strategies are inaccurate by several orders of magnitude. Specifically, the average estimation error for average-bytecode at the whole program level was 133%, and for no-path-sensitivity was 267%. To understand why average-bytecode is inaccurate, we plotted a distribution of bytecode energy costs. This distribution, omitted for brevity, is highly skewed, with a small number of instructions using more than an order of magnitude more energy than the rest. Thus, an average bytecode cost can either inflate the energy estimate for a program that does not use these expensive instructions, or underestimate energy usage for programs that do. Moreover, our results also indicate that path-sensitivity is crucial for capturing energy usage of applications and methods. This is because our subject applications are significantly large, and have a large number of potential paths that may be explored during an execution.

Overall, these results provide a compelling demonstration of the accuracy of *eLens*, its ability to provide energy estimates at granularity that is beyond the reach of hardware power monitors, and of the importance of the two pillars of its design (per-instruction energy modeling and path sensitivity).

C. Why Do We Need Energy Profilers?

Traditionally, using execution times obtained from a method profiler is a common way for developers to identify methods they will focus on to improve the efficiency of their application. In this section, we show that execution time may not be a good proxy to identify energy-inefficient segments of code and that specialized energy profilers like *eLens* are necessary. To demonstrate this, we first calculated the execution time and energy estimate of each method of each application. To obtain the execution time we profiled each method using timestamps and calculated the corresponding energy estimate for the method using *eLens*. We then compared the information in two different ways to evaluate whether time is, indeed, a reasonable proxy for energy cost.

Correlation: We first determined whether there is a linear correlation between the execution time of a method and its



(a) Whole program accuracy (b) Method-level accuracy (c) Top five energy hotspots

Fig. 3: Plots of *eLens* accuracy comparison and top energy hotspots

estimated total energy (across all components). We do this by calculating the Pearson correlation coefficient of the two series. Values of the coefficient closer to 1 or -1 indicate that the two series have a strong (positive or negative) linear relationship and values closer to zero indicate that they are uncorrelated. The Pearson coefficients (r in Table I) are nearly zero across all applications, *indicating that there is almost no linear correlation between execution time and energy usage*.

Ranking similarity: We then considered that even if energy and time were not linearly related, the relative rankings by the metric might provide useful guidance. So we measured the similarity of the rankings by calculating their cosine similarity, a technique used to measure similarity between two vectors in n-dimensional space. In this case, we defined two vectors, $v_0, v_1, \ldots v_{|methods|}$, where each v_i was defined for one vector as method i's energy rank and for the other, method i's execution time rank. The cosine similarity ranges from -1 to 1, with -1 denoting the exact opposite ranking, 0 denoting independent rankings, and 1 denoting the same ranking. Table I shows cosine similarity values closer to 0 than to -1 or 1 for all but one application, Skyfire. This strongly suggests that, for most applications, time and energy are almost independent.

There are at least two reasons why time and energy are uncorrelated. The first is that many hardware components have multiple power states. Two different methods can take the same time to execute on a CPU, but when one of them is executing, the CPU may be at frequency f_1, while when the other is executing it may be at f_2. If $f_2 > f_1$, the energy consumed by the latter will be more than that consumed by the former. The second explanation lies in the asynchronous design of system and API calls. When an application sends data over the network, that data is buffered by the operating system so the application may not be charged the time taken to transmit the data. However, *eLens* will accurately account for the energy cost of transmitting the data, since it profiles the send API call.

These results demonstrate the importance of an approach like *eLens*, which can guide energy optimizations more accurately than method profilers based on execution time.

D. Analysis Time

In this section, we evaluate one aspect of usability of *eLens*, namely whether it is fast enough to be used during software development. We measured for each application, over a series of executions, the time the Workload Generator took to instrument (T_{Inst}) and the time needed by the Analyzer to calculate the energy estimate given the output of the Workload Generator (T_{est}). Table I shows these two measures, for each application. As the results show, the instrumentation time ranged from about 4 to 14 minutes and the analysis time ranged from 6 to 17 seconds. In practice, instrumentation time would be much lower because after each developer iteration, it would only be necessary to reinstrument the changed classes as opposed to all classes (our numbers report the latter). The analysis time is fairly low in comparison and would not hinder usability during development.

We did not measure the overhead introduced at runtime by our instrumentation because our method of executing use cases was to manually interact with the application and we could not control for normal human variations when interacting with the instrumented and non-instrumented versions of the application. Anecdotally, the runtime overhead was imperceptible to users interacting with the instrumented application. Using the additional estimated energy consumed by our instrumentation as a proxy, we estimate that the runtime overhead ranged from 0.2%-7.2% across the applications.

E. Using eLens *to Compare Applications*

We conclude by illustrating how *eLens* can be used to compare application energy usage. We have left a more complete study of application energy characteristics to future work, but this section begins to explore how *eLens* can be used to understand where energy is expended in an application, and how that differs across applications.

To study application energy usage, we calculated, for each application, the top 5 energy hotspots. The bar graph of Figure 3(c) plots the subject applications on the x-axis, and, for each application, the fraction of total application energy consumed by the top 5 hotspots, in order from left to right. As this figure shows, applications vary widely in how energy usage is distributed. The fraction of total energy that can be attributed to the top 5 hotspots varies from about 10% for *Bubble Blaster* to an astounding 85% for *Skyfire*. Moreover, as the figure shows, even among the top 5 hotspots, energy consumption can be significantly skewed; in *Skyfire*, a single hotspot uses over 80% of application energy!

Less evident from this figure is an understanding of the causes of high energy usage, but an examination of the top 5 hotspots (listing omitted for space reasons) reveals interesting insights. In *Skyfire*, an Android library call for HTTP downloads consumes the most energy, but in the *BBC*

Reader, the top 5 hotspots include API calls for downloading and rendering content as well as downloading advertisements. In *Textgram* as well as in the two games, appropriately enough, graphics computations dominate the top 5 hotspots. In these cases, an included package or library consumed a significant amount of energy; in response to this, an application developer might either choose to optimize the library implementation or re-implement the functionality provided by the library in a more energy-efficient manner.

VI. Conclusion

This paper presents a new technique, *eLens*, for estimating energy consumption of applications written for Android mobile devices. *eLens* brings together two ideas, per-instruction energy modeling, and program analysis, in order to accurately, and without requiring power measurement hardware, estimate application energy usage at the level of granularity of the whole application, method, path, or source code line. An evaluation of *eLens* on six marketplace applications shows that its energy estimates are accurate to within 10%, and its run time is acceptable. Moreover, *eLens* can reveal insights about energy usage across different applications, and its energy estimates are uncorrelated with execution time, suggesting that method profilers may not help in optimizing applications for energy use. Overall, the results of the evaluation were very positive and indicate that *eLens* is an accurate, fast, and useful technique for estimating energy consumption.

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. Any opinions, findings, conclusions, or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the sponsors. The authors would like to thank Nilesh Mishra for his help on the LEAP platform. The first author was supported by Annenberg Graduate Fellowship.

References

[1] MonkeyRunner. http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html.

[2] Robotium. http://code.google.com/p/robotium/.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] S. Hao, D. Li, W. G. Halfond, and R. Govindan. Estimating Android Applications' CPU Energy Usage via Bytecode Profiling. In *First International Workshop on Green and Sustainable Software (GREENS)*, pages 1–7, 2012.

[11] 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.

[12] 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.

[13] H. Mehta, R. Owens, and M. Irwin. Instruction Level Power Profiling. In *Proc. of Acoustics, Speech, and Signal Processing (ICASSP-96)*, volume 6, pages 3326–3329. IEEE, 1996.

[14] R. Mittal, A. Kansal, and R. Chandra. Empowering Developers to Estimate App Energy Consumption. In *Proc. of MobiCom*, pages 317–328. ACM, 2012.

[15] T. Mudge, T. Austin, and D. Grunwald. The reference manual for the Sim-Panalyzer version 2.0. http://www.eecs.umich.edu/~panalyzer.

[16] 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.

[17] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof. In *Proc. of EuroSys*, pages 29–42, 2012.

[18] 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.

[19] 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.

[20] C. Seo, S. Malek, and N. Medvidovic. An Energy Consumption Framework for Distributed Java-Based Systems. In *Proc. IEEE/ACM ASE*, pages 421–424. ACM, 2007.

[21] 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.

[22] 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.

[23] A. Sinha and A. Chandrakasan. Jouletrack-A Web Based Tool for Software Energy Profiling. In *Proc. of Design Automation Conference (DAC)*, pages 220–225. IEEE, 2001.

[24] 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*, 2001.

[25] J. Stoess, C. Lang, and F. Bellosa. Energy Management for Hypervisor-Based Virtual Machines. In *USENIX Annual Technical Conference (ATC)*, page 1. USENIX Association, 2007.

[26] 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.

[27] 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.

[28] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, 1994.

[29] 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.

[30] 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.

[31] 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.