Estimating Android Applications' CPU Energy Usage via Bytecode Profiling

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

***Abstract*—Optimizing the energy efficiency of mobile applications can greatly increase user satisfaction. However, developers lack easily applied tools for estimating the energy consumption of their applications. This paper proposes a new approach, *eCalc*, that is lightweight in terms of its developer requirements and provides code-level estimates of energy consumption. The approach achieves this using estimation techniques based on program analysis of the mobile application. In evaluation, *eCalc* is able to estimate energy consumption within 9.5% of the ground truth for a set of mobile applications. Additionally, *eCalc* provides useful and meaningful feedback to the developer that helps to characterize energy consumption of the application.**

***Keywords*-Android apps, bytecode profiling, eCalc, energy estimation**

I. Introduction

The power of mobile computation has exploded. Smartphones and tablets allow people to carry around more computational power in their hand now than most had on their desktop just a few years ago. However, the usability of these devices is strongly defined by the energy consumption of mobile applications. Despite careful operating systems design and improvements in hardware energy-efficiency, the battery lifetime of these devices can be adversely affected by poor application design.

Optimizing mobile applications for energy-efficiency can greatly increase overall user satisfaction. However, this optimization can, in practice, be very challenging and is often neglected. Techniques exist to help developers, but they have limitations which limit their usefulness. Cycle-accurate simulators are difficult to create for every hardware/OS combination and tend to run several thousand times slower than the actual hardware. Approaches that estimate hardware component-level energy usage, such as disk and CPU state, do not provide enough information to tie estimates to the structure of the code. Developers could use field measurements, but especially among the mobile application developer community, very few development groups possess the infrastructure, funding, or expertise to accurately perform this type of experimentation for each application they develop. These limitations leave developers without tools that can accurately and easily estimate software power consumption. So, they follow well-known best practices that provide general advice and guidelines; these can help developers make educated choices about their implementation decisions, but are not rigorous or objective ways to estimate power consumption.

To address these limitations, the authors propose a new technique, *eCalc*, that achieves lightweight fine-grained estimates of application energy usage. This technique is based on analyzing the execution traces of the application under development and using carefully developed cost functions to estimate energy costs that would be incurred by the instructions executed in the trace. The proposed technique is lightweight in that it does not require expensive energy monitoring equipment, instrumenting the underlying virtual machine and/or operating system, and can be easily integrated into development environments to provide energy consumption feedback during code development. Since the technique builds the estimate based on the execution trace, it can also directly correlate energy consumption with the structure of the code and provide fine-grained feedback to the developer. Our evaluation of *eCalc* shows that its energy estimates are within 9.5% of ground truth measurements obtained using a sophisticated power monitor and can be provided with a low runtime cost.

The paper is organized as follows. In Section II, we discuss the design of *eCalc*. Implementation issues related to profiling the software environment are discussed in Section III. Section IV contains the evaluation results of the approach. Related work is described in Section V and the paper concludes in Section VI.

II. The Design of *eCalc*

The goal of our work is to accurately estimate the energy consumption of a software artifact in a way that does not require power monitoring hardware and can be tied to the structure of the code. To do this, we developed a technique, *eCalc*, which uses program analysis to analyze an execution trace and estimate the amount of energy a software artifact would consume during that execution.

Figure 1 illustrates our approach. Items outside of the dotted line represent inputs and outputs of *eCalc*. The inputs to the system are: (1) the software artifact; (2) the test cases, which are used to generate the execution trace; and (3) the CPU Profile, which provides the energy cost function, $C(i)$, for each instruction i. Such a profile would typically be provided by the corresponding hardware manufacturers and/or developers of the operating system on which the

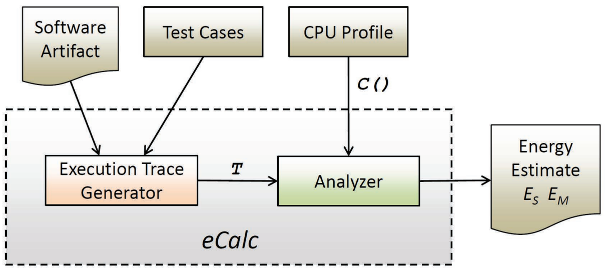

Figure 1: Overview of *eCalc*.

software artifact will be run. However, since this is not yet standard practice, Section III describes how a CPU Profile was created for *eCalc*.

The output of *eCalc* is an energy estimate for the software at either the method or whole program granularity. Within *eCalc*, there are two components: Execution Trace Generator and Analyzer. The Execution Trace Generator translates test cases into set of execution traces through the software artifact (II-A). The Analyzer uses the traces and the cost functions (II-C) to compute an energy estimate at each granularity level (II-B).

A. Obtaining Execution Traces

To estimate the energy consumed by a software artifact, it is necessary to make assumptions about which parts of the program will be executed and how often they will be executed. This is because energy consumption is not uniform over different paths of execution through the program. The *eCalc* technique estimates the power consumption of a single execution of a software artifact. To obtain the trace, *eCalc* instruments the program to be analyzed so that when it is run, the sequence of executed statements is captured. After a program has been instrumented, the developer runs the test cases for which energy consumption is to be estimated. These tests can be run either manually or automatically with tools such as MonkeyRunner[1], which provide a scripting capability for Android based applications. Since only the execution trace is needed, the program can be run on a simulator or actual mobile device. During execution, the instrumented version of the program behaves as normal, but also records information about the paths executed. This information includes the specific instructions executed and their location (*e.g.,* bytecode position) in the artifact. These execution traces are then used as input for the next phase of the *eCalc* technique.

B. Estimating Energy Consumption

Energy consumption estimation is performed by the Analyzer, which uses both the execution trace information (T) and the CPU Profile. *eCalc* provides estimates at both the software artifact (E_S) and method (E_M) level. The basic methodology is to identify the instructions in an execution that are grouped together at each level of granularity. Then the cost of each of these instructions is summed and the sum is associated with the grouping. The cost of each instruction i is given by a set of cost functions, $C(i)$, which are described in more detail in Section II-C. Using this methodology, it is also possible to obtain energy estimates at the source line level. The information necessary to do this is maintained in many compilers (*e.g.,* `javac`) by enabling the debug flag. However, defining and evaluating this level of granularity is deferred at this time for future work.

The calculation for each level of granularity is shown in Table I. To calculate the energy consumed by a given method m, the Analyzer first computes a function $Q(m)$, which returns all instructions that appeared in the execution trace and are in method m. Finally, the energy consumption at the software artifact level (E_S) is simply the sum of the cost of all instructions in the execution trace T.

Table I: Formulae for calculating energy consumption at different granularities.

Method	$E_M(m) = \sum_{i \in Q(m)} C(i)$
Software Artifact	$E_S = \sum_{i \in T} C(i)$

C. Defining Cost Functions

To calculate the potential cost of each individual instruction, *eCalc* relies on a set of energy cost functions denoted as $C()$. The way in which *eCalc* calculates cost depends on the type of instruction, as shown in Table II. Below, each of the categories is explained in more detail. Concrete values for the cost functions are obtained using the method described in Section III.

Standard Instructions All instructions that do not fit into one of the special categories listed below are considered to be standard instructions. Java-based examples include `iadd`, `bipush`, and `ldc`. For each of these instructions, the energy cost represents the measured cost of the instruction in the software environment. The function $C_M(i)$ returns the measured energy cost for a specific instruction i.

Invocations Instructions that call a method are considered invocations. Java-based examples are `invokevirtual` and `invokestatic`. In this approach, invocations are handled differently based on whether they call a method defined within application code or external libraries provided by the software environment (*e.g.,* `android.*` or `java.*`). For invocations that target application methods, the cost of the invocation is simply the overhead cost of an invoke. This is represented as a fixed cost C_I for all types of invokes. No other adjustments need to be made since the target code of the invocation will also be accounted for by *eCalc*. For calls to methods defined in the software environment, the approach uses models of the energy cost of the target method. These models are built per-method using profiling and estimation techniques. More details are provided in

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

Section III. The cost for a software environment method is represented as $C_{Sys}(i)$ and takes the instruction as input.

Returns Instructions that cause a method call to exit and return to the originating call site are considered returns. These are assigned a measured cost, which depends on the type of return (*e.g.,* `freturn` or `areturn`). This cost is represented as $C_R(i)$.

Allocations Instructions that cause memory to be set aside for an array or object are considered allocations. Java-based examples include `new` and `anewarray`. The cost function for these instructions is represented as $C_A(i)$ and takes the instruction as input. The function $C_A(i)$ attempts to estimate the energy cost of the allocation based on the size and number of objects to be allocated. These numbers are estimated using a "best-effort" static analysis of the instructions preceding the allocation statement in the path.

Table II: $C()$ for different instruction types.

$C_{Sys}(i)$	if i is an invoke and target(i) is a system call
C_I	if i is an invoke and target(i) is not a system call
$C_R(i)$	if i is a return
$C_A(i)$	if i is an allocation
$C_M(i)$	otherwise

At this stage of the research, we have focused on estimating CPU energy consumption only. In many mobile platforms, CPU energy usage is a significant component of the overall energy consumption, generally higher than memory or persistent storage and comparable to the wireless network. To get an accurate picture of energy usage, it is necessary to estimate these other components; however, in preliminary work, we have found that they require different forms of analysis and profiling than those discussed in this paper, so we have left these to future work.

III. CPU Profile

The CPU Profile (CPUP) defines energy cost functions for each type of instruction. In general, we anticipate that developers of hardware and software platforms will provide these values as part of a software development kit. However, currently it is not common practice to provide these. Therefore, we discuss the techniques we used to estimate the cost functions, $C()$. Note that this process has to be performed only once for a given software environment and the resulting profile can then be distributed as part of the software development kit provided to developers. This method of distribution makes it unnecessary for developers to use complicated or expensive energy monitoring equipment, which is one of the two goals of the proposed approach. Furthermore, by simply swapping CPUPs, developers can rapidly estimate energy consumption on different hardware/operating systems combinations.

The profiling process is performed by running individual instructions repeatedly over a period of time on the target hardware that is connected to a power monitor. There are several challenges: finding power monitors capable of fine-grained measurements; dealing with dependencies between instructions; and using special procedures for method invocations. We now describe how we address these challenges.

We have used the LEAP [12] power measurement device. LEAP contains an ATOM processor that runs Android-x86 version 2.x. Its analog-to-digital converter (or DAQ) samples CPU current draw at 10 KHz. 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 obtain 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 each pair of pulses.

To profile individual instructions, we created a set of test scripts each of which profiles an individual bytecode by placing the bytecode 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 in order to minimize the impact of cold starts, cache effects, or background processes (such as garbage collection).

Some instructions are dependent on the results of execution of other instructions. For example, an `iadd` 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 `ldc` is used to push operands on the stack, then the test script for `iadd` includes these bytecodes. When profiling `iadd`, we subtract the cost of the two `ldc` instructions. This process is repeated to build up scripts that can measure all of the Java bytecodes. For most instructions, the estimated power cost is the measured cost, C_M, derived from this process.

For `invoke` instructions, there are two possible scenarios: the target is an application level method or the target is a system call. For the former, the invocation execution will already have been instrumented by the Execution Trace Generator, and its energy cost included by the Analyzer, so the only cost associated with the instruction, C_I, is the measured energy cost of the invocation.

When the target is a system method, such as calls into any method whose signature begins with `java.*` or `android.*`, the CPUP defines a *model* of that invocation's cost. This model specifies the energy cost of the invocation C_{Sys} as a function of some feature of the invoke parameters (e.g., size). To calculate this model, we have written test scripts that calculate the energy for different values of these features; we then use linear regression to build a model for the invocation. Rough estimates for parameter size are obtained by analyzing statements in the path that precede the instruction.

The cost of `return` instructions is determined by measuring each return type in a pairing with an `invoke` instruction. This method was used because it was not possible to isolate invokes and returns; both are required to execute in pairs. In each case, the energy cost of the `return` instructions is calculated as half the cost of the `invoke-return` pair. Our measurements indicate that different types of `return` instructions (*e.g.,* `freturn` and `areturn`) varied in their energy cost, so C_R depends on the return type.

Instructions, such as `new` and `anewarray`, that cause memory to be set aside for an array or object, are considered allocations. Our measurements indicate that the cost of allocation statements is proportional to the amount of memory being allocated. Therefore, the cost of an allocation is a linear function with respect to the dimensions of the object or primitive being allocated. For estimating the dimension, we use a "best effort" analysis that examines the instructions in the path preceding the instruction and attempts to estimate the size. For example, an `ldc` instruction that immediately precedes an `anewarray` instruction specifies the array dimensions.

IV. Evaluation

In this section, we evaluate the *accuracy* and *usefulness* of *eCalc*. We begin with a brief description of the *eCalc* implementation, discuss the subject applications used in the studies, and then report the results of several experiments designed to evaluate these two aspects of *eCalc*.

A. Implementation

For the purposes of the evaluation, we implemented the approach in a prototype tool for the Android 2.x platform. To perform instrumentation, *eCalc* uses the Java Byte-Code Engineering Library (BCEL) (http://commons.apache.org/bcel/). The instrumented Java class files are then converted to Dalvik bytecode using the `dx` tool provided with the Android SDK. As discussed before, the CPUP relies on the LEAP platform to extract high frequency measurements of CPU energy usage. Using this, we profiled all the standard instructions, returns, invocations and allocations. For invocations that target system methods, only those needed for the evaluation were measured.

B. Subject Applications

In the selection of subject applications, we were primarily constrained by the need for repeatability of the experiments. Since most Android applications require user input via a graphical interface, it is hard to generate consistent workloads because user activities may differ subtly between different runs. For this reason, we chose five benchmark applications, divided into two classes, that permit repeatability.

The first class of benchmark applications contains only application code and does not invoke the system environment. These applications are publicly available and were not authored by us:

- `qsort`[2] performs a quick sort on a given 10K input array.
- `md5`[3] computes the digest of a 2MB memory chunk.
- `matrix`[4] solves a 200x200 matrix using Gaussian Elimination. This benchmark comes from the standard `linpack` suite of benchmarks.

While these benchmarks are compute-intensive, they each use qualitatively different mathematical and logical instructions. However, these benchmarks only use standard bytecodes. To demonstrate that *eCalc* is accurate for applications that invoke the system environment, we constructed two other benchmark applications.

- `imgrot` rotates an 1024x1024 input image by invoking `sin` and `cos` API functions in `java.lang.Math`. The energy usage of these functions does not depend on input size, so we profiled them in a manner similar to standard instructions.
- `md5api` copies in a 10MB buffer and computes the digest in 2KB chunks by invoking the `update` and `digest` API functions in `java.security.MessageDigest`. These API functions' energy usage depends upon the size of the input, and we constructed linear models for their energy usage.

C. *Accuracy of* eCalc

In this section we evaluate the accuracy of *eCalc* for estimating energy consumption. This is done by comparing the *ground-truth* energy usage of an application to the *eCalc* estimate.

Metrics and Methodology. The ground truth can be computed by taking hardware measurements on our power monitor, but there are two challenges in getting accurate results. First, any hardware power monitor (including the LEAP) cannot isolate the energy usage of a specific application on a multitasking system like Android; rather, it measures the aggregate energy usage of the CPU, which includes system activity (background daemons, scheduler, etc.) as well as background virtual machine activity (*e.g.,* garbage collection). These can pollute any ground-truth measurement, including the CPUP.

Second, hardware power monitors measure energy consumption by sampling the voltage or current to the corresponding component (e.g., the CPU). The sampling frequency determines the granularity of energy measurement

[2] http://code.google.com/p/android-benchmarks
[3] http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html
[4] http://www.netlib.org/benchmark/linpackjava

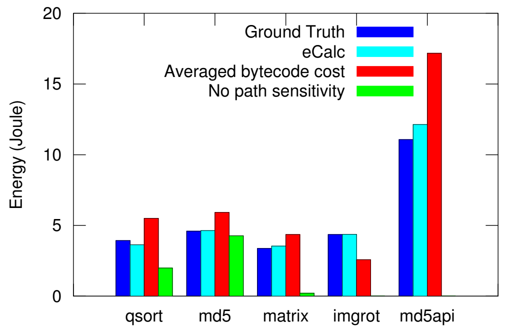

Figure 2: Accuracy comparison between Ground Truth (GT) and *eCalc* estimate over 5 benchmark apps.

and limits the kind of accuracy validation we are able to do. The LEAP samples at 10 KHz, so in theory it can capture energy usage of functions that last for at least 0.1 ms. In practice, we have found that reliable energy estimates can only be obtained for functions which run for at least 10 ms.

Note that these issues do not affect our approach, only the ground truth measurements. *eCalc* is able to estimate energy usage of arbitrarily small functions because it uses profiled costs of bytecode energy usage. For the same reason, *eCalc* can isolate the energy usage of application code from background energy consumption. These are two significant advantages of our approach.

To overcome the first challenge, application isolation, we turned off all unnecessary background services and applications. In addition, for the subject application that we want to measure ground truth data, we used the Linux `nice` command to increase its scheduling priority so that the chance of being preempted by the scheduler is minimized. However, there still exists low probability our app being swapped out. To deal with such uncertainty, we conducted several runs of ground truth measurements and picked the smallest value as the representative ground truth. The second challenge, sampling frequency, constrains us to demonstrate accuracy only at the software artifact level and for methods that run for a relatively long time.

For each of these applications, we obtained the ground-truth estimates as described above. We also ran each application once under *eCalc* to extract the execution trace and compute energy usage at the software artifact and method level.

Accuracy at the Software Artifact Level. Figure 2 compares *eCalc* estimates against ground-truth measurements. *In all cases,* eCalc *estimates are within 9.5% of ground-truth.* This is a strong result that validates the *eCalc* approach and suggests that accurate energy estimates can be provided to developers during the development cycle.

The discrepancy is highest for the `md5api`, 9.5%. We expect that *eCalc* estimates for this application can be improved by a more careful profiling of the `update()` and `digest()` API calls. Our current linear model is generated by computing the energy for a small number of input sizes; increasing this sample set will likely improve the accuracy of the estimation.

Accuracy at Method Level. To quantify the accuracy of *eCalc*'s method level energy estimates, we use the three benchmark applications `qsort`, `md5` and `matrix`. As discussed before, even at its high sampling rate, LEAP does not have a high enough resolution to calculate ground truth for all methods in our benchmark. Based on an examination of each benchmark, we selected a few likely long-running methods in each for which we thought we might be able to obtain ground-truth energy usage.

Table III: Method-level accuracy: Ground Truth obtained by LEAP vs. *eCalc*.

App	Ground Truth	*eCalc*	Accuracy Gap
qsort	1/3	3/3	8.98%
md5	1/7	7/7	1.05%
matrix	1/10	10/10	4.85%

Table III shows *eCalc*'s ability to obtain energy estimation at the method level and its accuracy gap[5] against the ground truth values provided by the LEAP power monitor. For `qsort`, we chose three methods, but LEAP was able to measure ground truth for only one of these. For that one method, `sort()`, *eCalc* estimate was within 9% of ground-truth. Significantly, *eCalc* was able to provide energy estimates for the other two methods for which LEAP could not measure energy consumption. This illustrates the power of our approach, which can estimate energy consumption at much finer granularities than possible with a fairly sophisticated power monitor.

Similarly, for our two other benchmarks `md5` and `matrix`, we instrumented 7 and 10 methods respectively, but LEAP was only able to calculate ground truth for one. In `md5`, for the only method that LEAP can measure its energy cost, `update()`, *eCalc* estimate was within 1.1% of ground truth and for the only method in `matrix`, *eCalc* estimate was off by less than 5%.

Overall, these results provide a compelling demonstration of the accuracy of *eCalc* even at the method level, and its ability to provide energy estimates at granularity that is beyond the reach of hardware power monitors. As we have discussed in Section III, there are several ways in which we can improve the accuracy of *eCalc*, so these numbers are likely an upper bound on *eCalc*'s error.

D. *Usefulness of* eCalc

In this evaluation, we evaluated usefulness by comparing *eCalc* to two heuristics for estimating energy consumption. The first is to assign the same averaged energy cost to all instructions instead of individual costs, as is done in

[5]Defined as $\frac{|eCalc - \mathrm{GT}|}{\mathrm{GT}}$.

our approach. However, energy costs can vary significantly across bytecodes, which means that this simpler strategy is unlikely to yield accurate results. To illustrate, Figure 2 also shows the impact of replacing each bytecode energy cost with the *average* value. For a couple of the benchmarks, the estimates come close to the *eCalc* estimates, but are significantly off for the others (e.g., almost 50% off for the `md5api`).

A second alternative is to measure how many times a method is called and estimate each method's energy cost as the sum of all the bytecodes in the method definition, regardless of the execution path it took during runtime. Figure 2 shows that this path-insensitive approach results in estimates that are very inaccurate even when the profiled bytecode costs are accurate. For only one of the applications, `md5`, does this method come close and that is because the application consists of straight-line code. For all others, which have more complicated program structures with nested loops within methods, this approach is extremely inaccurate; for three of the applications, the estimate was less than 1% of the ground truth.

Lastly, measurements of the runtime of the *eCalc* approach during this experimentation showed that the combined maximum runtime for instrumenting and estimating any of the subject applications was three seconds and the majority of the runtimes were under one second. However, since these results were for small benchmark applications, further experimentation with larger and more realistic applications is needed to confirm this observation.

V. Related Work

Power modeling is a broad area of research that encompasses several sub-fields, including architecture, operating systems, and software engineering. To our knowledge, we are unaware of work that has attempted bytecode level profiling for estimating fine-grained power usage. Given space limitations, we present representative pieces of work from each area to place *eCalc* in the context of prior work.

Closely related to *eCalc* work 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 [19][20] model the energy of an instruction using a base energy as well as a transition energy for each pair of instructions. In contrast to this body of work, our approach estimates bytecode energy costs; energy of bytecodes shows considerably higher variation and may be less affected by architectural effects between instructions.

Cycle-accurate simulators have also been developed to estimate software energy consumption (such as Sim-Panalyzer [9] and Wattch [5]). These approaches can simulate the actions of a processor at architecture level and estimate energy consumption in each cycle. Compared to *eCalc*, these methods can be highly inefficient (e.g. Sim-Panalyzer needs 4300 instructions to simulate the execution of one single instruction) impeding their usability for complex mobile applications that involve user interaction.

Other works focus on the energy consumption of the operating system at a routine or system-call level [6], [8], [17], [22]. They all build power models at the system or routine call level; a power model describes 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 *profile* system-level energy consumption during the execution of the targeted program. Our approach borrows some of this methodology to model method invocations, but, unlike this body of work, *eCalc* is able to *predict* power without monitoring it during runtime.

A complementary approach [10] explicitly models the state transitions between hardware power states of smartphone hardware components (CPU, memory, WiFi 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 which invoke system calls. However, unlike *eCalc*, this requires manual instrumentation of the application and may not work for applications with energy-intensive application level code.

Beyond instruction-level and call-level energy modeling, some work has also considered *path* energy profiling. Tan and colleagues [18] model path energy costs using the Ball-Larus profiling technique [3]. By contrast, *eCalc* directly estimates bytecode costs and environment invocations, so can be used to compute energy consumption at different granularities.

Like *eCalc*, Seo and colleagues [13], [14], [15] built an instruction-level model for Java bytecodes and a linear model for interfaces of the JVM and operating system-calls. Unlike *eCalc*, 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 whole-program level.

Complementary methods for estimating the energy usage of applications have relied on operating system level instrumentation [7], [4], together with hardware support for energy measurement. Xian and colleagues [21] proposed a programming abstraction to help developers make online energy-efficient decisions, which requires in-depth detailed knowledge about the application logic. Recently, Pathak and colleagues [11] developed the `eprof` tool by using system-call and routine tracing. PowerTutor [1] uses a utilization-based approach to profile energy cost, at runtime, at the granularity of application or software artifact level. These works focus on energy profiling for a specific hardware platform at runtime and will incur significant overhead to move to a different one. In contrast, *eCalc* only relies on power profiles and does not require modifications to the operating system. Hence, it is able to *predict* energy usage across different platforms for given execution traces of the application.

Finally, some work has explored the energy characterization of network protocols and devices [16], [2]; *eCalc* currently focuses on CPU energy consumption, but can use the results of this research to estimate network energy usage and more completely model energy consumption.

VI. Conclusion

This paper presents a new technique, *eCalc*, for estimating energy consumption of applications written for Android mobile devices. Previous approaches either required expensive energy monitoring equipment or could only provide estimates at the whole-program level. *eCalc* does not require monitoring equipment and can provide energy estimates at the level of the whole program, method, or source line. *eCalc* was implemented and evaluated for accuracy and usefulness. For the set of subject applications, *eCalc* was accurate within 0.14% to 9.5%. The evaluation also showed that *eCalc* was useful for developers, computing results that were more accurate than commonly used proxy techniques. Overall, the results indicate that *eCalc* is an accurate, fast, and useful technique for estimating energy consumption.

References

[1] PowerTutor. http://powertutor.org.

[2] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *Proc. of IMC*, pages 280–293. ACM, 2009.

[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] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] H. Singh, S. Saxena, and S. Singh. Energy Consumption of TCP in Ad Hoc Networks. *Wireless Networks*, 10(5):531–542, 2004.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] C. Xian, Y.-H. Lu, and Z. Li. A programming environment with runtime energy characterization for energy-aware applications. In *Proc. of the 2007 International Symposium on Low Power Electronics and Design (ISLPED)*, pages 141–146. ACM, 2007.

[22] 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.