Survey of Approaches for Assessing Software Energy Consumption

Felix Rieger
Philipps-Universität Marburg, Germany
riegerf@mathematik.uni-marburg.de

Christoph Bockisch
Philipps-Universität Marburg, Germany
bockisch@acm.org

Abstract

The energy consumption of software-controlled ICT systems ranging from mobile devices to data centers is increasingly gaining attention, however, energy optimization is still far from an established task in the software development process. Therefore, we have surveyed the available research on assessing the energy consumption of software systems, which showed a lack of development tools, but several approaches exist for measuring the energy consumption. We group these approaches according to how measurement data is made available and compare several characteristics of the collected data. The survey shows that not only development tools for software energy optimization are still missing, but there is also a lack of fine-grained measurement approaches as well as approaches for general-purpose platforms.

CCS Concepts • **Hardware** → *Energy metering*; • **Software and its engineering** → *Monitors*; *Power management*;

Keywords Survey, Software Energy Behavior, Energy Measurement, Energy Analysis

ACM Reference Format:
Felix Rieger and Christoph Bockisch. 2017. Survey of Approaches for Assessing Software Energy Consumption. In *Proceedings of ACM SIGPLAN International Workshop on Comprehension of Complex Systems (CoCos'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3141842.3141846

1 Introduction

Comprehending the energy consumption of software-intensive systems is increasingly important, mainly for two reasons. First, the usefulness of mobile devices like smart phones or wireless sensors is heavily influenced by the battery life. Second, the energy consumption of ICT (Information and Communication Technology) systems is becoming a growing environmental and economical factor. For example, the SMARTer 2030 report [4] estimates that all ICT systems world-wide combined will make up for 2% of the global carbon emission in 2030. While the interest of developers in the energy consumption of their software increases, recent empirical studies have shown that awareness is still very low [14] and misconceptions are very frequent [17].

For this reason, we want to evaluate the current state of the art of tools that support programmers in understanding their programs' energy behavior. Our main interest is in directly usable tools for programmers. We followed related work and references that seemed to imply tooling from the literature we encountered in previous work about energy measurement hardware and software. We also carried out a quick web search to find (commercially) available tools.

2 Survey

To our surprise, there are only few approaches that can be applied to custom code and that provide feedback linked to this code. For example, information could be presented as an overlay in the source code editor or in a dedicated view linked to the code. Those approaches that we identified are presented in Section 2.1

Since there are very few *tools* available supporting *comprehension* of programs' energy behavior, we shift the focus of our survey to categorizing *approaches* for *assessing* programs' energy behavior. Such approaches form a first step in the comprehension task and can possibly lead to tools like we have sought in the beginning. In Section 2.2, we present these approaches and compare their utility with respect to building tools based on them.

In our literature study, we identified another category of approaches concerned with considering energy behavior in software engineering, namely best practices for writing energy-efficient code. Such approaches are mainly based on known usage patterns of certain libraries that lead to suboptimal energy behavior. While these works do not directly meet our objective, as they only identify predefined cases, they are still noteworthy in the context of this survey and are presented in Section 2.3.

2.1 Tools

Silicon Labs [19] provide an Eclipse-based IDE and development boards for ARM Cortex microcontrollers that allow

CoCos'17, October 23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5521-6/17/10…$15.00
https://doi.org/10.1145/3141842.3141846

measuring and profiling the energy consumption of embedded software at runtime. Energy consumption measurements for each method call are provided in a table. This system is commercially available and uses a low-cost development board with fast current and voltage sensors. The processor's program counter is read periodically and its value is sent to the host computer along with voltage and current measurements. The program running on the microcontroller is statically linked, uploaded by the host computer and resides at a previously known and constant address in memory. Hence, the address of the program counter is sufficient to infer the currently executed method.

Green Droid [3] instruments Android programs and uses PowerTutor [24] to measure their energy consumption at runtime. A trace of the instrumented program and the measured energy consumption are then used to create visualizations in the form of diagrams. These visualizations can help programmers to identify parts of their applications with problematic energy behaviour. To measure energy consumption, PowerTutor relies on a component-based model of the device's energy usage that can be automatically generated by running a calibration program on the device under test.

ELens [10] annotates Java source code for Android with energy consumption estimates. An IDE visualizes this by coloring code according to its estimated energy consumption. The estimates are derived from an instruction-level energy consumption model with path dependencies that has been created by the authors for some specific device. Evaluating this approach with actual energy measurements gives an accuracy error of about 10% for methods that run longer than 10ms. Methods with a runtime of less than 10ms could not be measured by the setup used by the authors.

To summarize, the tools that are currently available focus on a few specialized platforms. To our knowledge, there are no ready-to-use tools focusing on understanding energy consumption of general-purpose programs on a method level.

2.2 Measuring

We split the approaches that we investigate into three groups, which are discussed in the following:

- **without model derivation**: Approaches mainly concerned with a setup for measuring the energy consumption of one program run and providing this data with little or no abstraction.
- **with model derivation**: Approaches focusing on deriving reusable models of energy consumption from measured energy behavior to later predict the energy behavior of a program without actually measuring it.
- **power analysis attacks**: Approaches in this group stem from the security domain and are concerned with security attacks based on the measurement and analysis of energy consumption. Their methods can possibly be transferred to our objectives.

2.2.1 Without Model Derivation

These approaches directly measure energy consumption of the running program and use it for later analysis. They do not use it as input for models describing the energy consumption. This approach often requires an intricate measurement setup for the device the program will be running on. In our study, we identified a few features that are suitable for the comparison of such approaches:

- *Architecture*: An approach can either target a special-purpose or general-purpose architecture.
- *Granularity*: The approaches differ in the granularity at which they can measure.
- *Assumptions*: The assumptions that approaches makes about the measured program or its environment.

Silicon Labs [19] provide a commercially-available development system. *Architecture:* It is available for ARM Cortex-M microcontrollers running at up to 72 MHz that consists of a development board with integrated power measurement and data acquisition, and a corresponding IDE. *Granularity:* The IDE allows programmers to debug their program and obtain actual energy measurements for every method call. This run-time profiling has very little overhead, as the program running on the microcontroller is not instrumented. Instead, debugging hardware periodically reads the program counter register from the processor via an in-circuit debugging interface and sends it to the computer running the IDE along with energy measurements. *Assumptions:* This approach is generally not feasible for general-purpose programs running on general-purpose computers: There are no facilities for unobtrusively reading the PC register outside of the currently running program.

Ferreira et al. [7] describe the **SEFLab**. *Architecture:* They provide an approach for measuring software energy consumption in general-purpose x86 computers. The idea is to measure individual energy consumption of hardware components while running test scenarios on software. To this end, an x86 computer is outfitted with current and voltage sensors in the voltage supply lines leading to each component. *Granularity:* Energy consumption of software is measured using test scenarios. Software is executed in different test scenarios and energy consumption of each hardware component is measured, aggregated for each scenario. This paper provides the interesting insight that energy consumption can not always be reduced to CPU energy consumption. While this is not especially fine-grained, there is nothing speaking against applying this methodology to individual test cases of software. Measuring energy consumption per component could help understand the impact of software on different components of the computer and help steer programming decisions (e.g. writing a file to disk directly or first completely buffering it in memory and writing it to disk all at once).

JouleUnit [23] provides a framework for unit testing focused on energy testing. *Architecture:* JouleUnit is developed

to be an architecture-agnostic framework and thus reusable. It was evaluated using Android applications and applications on a robotics platform. It provides a framework consisting of a Coordinator, a Profiler, and a Workload Runner. The Coordinator initializes measurement and analyzes data. The Profiler takes the energy consumption measurements on the device. The Workload Runner runs the tests on the device. *Granularity:* JouleUnit executes all tests sequentially and collects timestamps of energy measurements and test executions. This also works across devices when the device running the test is not the device taking the measurements.

Understanding the history of the energy consumption of a piece of software can help to identify changes that introduced energy inefficiency. **Green Mining** [11] presents an approach that combines software repository mining with energy analysis. *Architecture:* This approach analyzes the energy consumption of software running on a general-purpose x86 PC. *Granularity:* Energy measurements are taken by running multiple versions of the software and executing a test scenario (e.g. loading websites or sending a file over the network), similar to SEFLab. Measurements of different versions can then be compared in order to detect changes in energy behavior. The energy consumption of the system is measured once per second at the power outlet before the main power supply. Unlike SEFLab, this approach does not measure the energy consumption contribution of individual hardware components. An idea of this work was to correlate software metrics and energy consumption, but findings were inconclusive. Automatically measuring the energy consumption of software versions in a repository could be valuable for a continuous-integration infrastructure, alerting developers early about potentially problematic energy behavior. However, whole-system energy consumption before the main power supply does not allow a lot of insight into what could possibly lead to changes in energy consumption.

Greendroid [3] is used to understand energy behavior of Android programs. *Architecture:* Unit tests of Android programs are instrumented with calls to an energy measurement API. When executing the tests, a call trace is generated along with energy measurements. These data are later analyzed and visualized. *Granularity:* Greendroid focuses on providing different visualizations of energy behavior. The provided visualizations are: execution time versus power consumption per second; power consumption by different hardware components during the test case; and power consumption related to source code. Rather than directly providing an energy measurement of a method execution (as is the case in [10, 19], Greendroid lists which methods were called when energy consumption was high. *Assumptions:* Greendroid uses PowerTutor [24] to measure energy consumption. Although PowerTutor relies on a device energy model to provide synthetic energy measurements, Greendroid does not infer further models relating power consumption to the source code.

2.2.2 With Model Derivation

These approaches measure energy consumption, but use the measurements to set parameters in an energy consumption model. This model is later used for analysis and prediction of program energy consumption. Having an energy model allows reasoning about the energy footprint of software before it is executed on a device. For the comparison of approaches in this group, we use the same criteria as in the previous section. Since a key characteristic of the approaches here is model building, we add as a criterion the **abstractness** of the derived model.

Economou et al. [5] present their idea to build a power consumption model of a general-purpose computer. *Architecture:* The approach was evaluated using an x86 server and an IA64 server. The power consumption model is constructed by correlating system utilization metrics to measured power consumption over a wide variety of applications. *Granularity:* The energy consumption for building the model was measured using a power meter between the power grid and the computer's power supply. The authors also instrumented the computer with custom power measurement hardware on the board level, but no further details were given. An accurate power consumption model based on performance counters and system utilization metrics allows predicting power consumption without dedicated measurement instrumentation. The prediction can be updated as quickly as system utilization metrics can be updated.

PowerTutor [24] uses a parameterized power consumption model. *Architecture:* The power consumption model is defined for Android smartphones and takes into account the state of different components (e.g. wireless networking, screen brightness, GPS status, etc.) that could potentially have an influence on power consumption. *Granularity:* The authors recognize differences in power consumption between different Android devices, even between different devices of the same model. To this end, an initial calibration phase stresses the phone's components individually to excite maximum power consumption. The measured power consumption of each component is then input into the model. When the power consumption of each component is known, the model can calculate the power consumption of the phone for various component states. The authors also describe a way to measure power consumption without specialized hardware sensors, using the battery voltage and discharge characteristics. This enables the approach to be run on any smartphone without first instrumenting it for current measurement. *Abstractness:* PowerTutor does not provide facilities for program analysis, dealing with whole-device energy consumption instead, but its model and infrastructure are used as measurement instrumentation for other work (e.g. [3]).

Work on estimating the **Energy Consumption in Pervasive Java-Based Systems** [18] proposes a model and

methodology for estimating a program's energy consumption at design time. *Architecture:* The authors use Java and the JVM running on handheld embedded devices. *Granularity:* To estimate the energy consumption of software, the software is decomposed into components. A component is a part of a program that serves a function. The energy consumption estimate relies on modeling energy consumption on the level of a component. Energy consumption is composed of the energy used for computation, communication, and an infrastructure overhead (mostly garbage collection). Communication overhead is dependent on the size of the exchanged message. *Abstractness:* To model computation power consumption, components are categorized into three types: energy usage (1) independent of input parameters, (2) linearly dependent on input parameter size, and (3) without direct relationship to input parameters.

Instruction Level Power Analysis And Optimization of Software [20] proposes to determine the amount of energy used for each CPU instruction. This serves as a basis for optimization of software with respect to power consumption, as power consumption of the software could be reliably predicted from its compiled code. *Architecture:* The authors demonstrate the applicability of their approach on three architectures: a general-purpose 486DX2 running at 40 MHz, a general-purpose 32-bit SPARC processor running at 20 Mhz, and a specialized DSP running at 40 MHz. *Granularity:* Instruction-level energy consumption is measured by executing the same instruction in an infinite loop and measuring the average power consumption of the system. Evaluation boards designed for current measurement and adapter board were used. The measured current was averaged over a 100 ms period. According to the authors, this approach is unsuitable for measuring large programs. *Abstractness:* The measured model includes power usage for each instruction. Power usage consists of the base cost, a circuit state overhead, and a cost for other-instruction effects. The base cost models the energy used by the instruction. The circuit state overhead describes the additional energy used by executing multiple different instructions (and not the same instruction in a loop). This is determined per instruction pair. The other-instruction effects model cache misses and other stalls. The authors find that "energy and running times track each other closely," proposing to optimize programs for speed as a first step to optimizing them for energy consumption. *Assumptions:* The approach of measuring the energy consumption of instructions by executing the same instruction in a loop assumes that an instruction's energy consumption is independent of the context the instruction is executed in. This problem is solved by measuring pairs of instructions to compute their overhead. The effect of stalls and cache misses are briefly mentioned and given a fixed energy consumption value. Since this paper is comparatively old, many features of modern processors are not taken into account. For example, modern processors have software-controllable dynamic frequency scaling and dynamic supply voltage [12], whereas the processors in the paper operate at a fixed clock frequency and supply voltage, therefore, we propose to re-evaluate this finding.

One step, proposed by **Eder et al.**, towards providing detailed energy analyses for whole systems is carrying out energy consumption analysis on the instruction level [6]. To this end, an energy model representing each processor instruction is generated. This energy model and the program to be analyzed are used as input to a resource analysis framework. *Architecture:* The authors build an energy model for the XMOS architecture. The XS1 architecture is a 32-bit hardware-multithreaded RISC architecture designed for embedded systems. In [6], the authors note that "the techniques we developed are readily transferable to other deeply-embedded, cache-less IoT-type processors such as those in the ARM Cortex M series or Atmel AVR." *Granularity:* The model is generated using measurements with custom measurement hardware. A master processor coordinates the execution of programs on the device under test. A subset of the ISA was characterized. The energy consumptions of non-characterized instructions are approximated using an average value. *Abstractness:* The model is based on the model described by [20]. However, it was adapted for the hardware multi-threaded architecture. The approach was evaluated by running benchmark programs, measuring their energy consumption and comparing them to the results computed by utilizing the model. Results for different input sizes were obtained. The difference between the computed energy consumption and the measured energy consumption is less than 20% most of the time, but was measured to be as high as 150%. *Assumptions:* The authors note that static analysis requires architecture-specific approaches, e.g., the presence of hardware multi-threading can have an impact on the energy consumption, even in the single-threaded case.

In [13], the **Li and Gallagher** use an intermediate energy model (consisting of *energy operations*) to map source code to energy consumption. *Architecture:* The approach was developed for an ARM-based single-board computer running Android. *Abstractness:* The source code of a program is split into blocks. A block has only one incoming and one outgoing edge. When the program is executed, block executions are traced. Blocks consist of energy operations that are grouped into classes (e.g. arithmetic, comparison, etc.). Energy operations are used as an abstraction layer for the model. Instrumenting the program with tracing instructions is seen as having an impact on the program's running time and energy consumption. This added instrumentation reduces sampling precision. Introducing blocks as energy operations should lessen the impact of tracing instrumentation on the program's running time and energy consumption. With the captured energy profile and energy operations, a model is

trained using regression analysis to infer correlations between energy operations and energy consumption from the captured data. This model can then be used to calculate energy consumption of source code. *Granularity:* The program under test is run twice. In the first run, tracing is enabled to record block execution. In the second run, an energy profile is captured. Measurements are taken using integrated current sensors of the single-board computer at 30 samples per second. The approach was evalated using a fixed test scenario where a game engine reacts to input and draws a moving sprite. Parameters of this test scenario were varied to generate a training and a test set. The authors report the inference error between training and test sets to be "within 10%". *Assumptions:* The model assumes that "the types of operation and operands provide sufficient information to estimate the average energy cost." [13]

2.2.3 Power Analysis Attacks

The domain of security research provides us with a different perspective on understanding the energy consumption of programs. Usually, energy consumption of a program varies according to the path taken in the program. Programs checking passwords or keys to authenticate a user will take different paths depending on whether a character in the password is correct or incorrect. Since different paths exhibit different energy behaviors, a password can be reconstructed from this information if accurate and fast energy measurements can be carried out when the program is provided with arbitrary passwords. These techniques are generally known as side-channel attacks and more specifically as power analyses.

More recent research has focused on extracting information from systems that do not guarantee exact repeatability of measurements. Implementations may obfuscate power usage by introducing clock jitter or other spurious signals either intentionally or as a byproduct of physical characteristics of the hardware and environment. As a consequence, counter-attacks have been developed to compensate the resulting distortions of the measurements. Using time-series analysis techniques such as wavelet transformation [2] or dynamic time warping [21] allows the construction of a single energy trace from multiple measurements.

Cryptographic ASICs on smart cards have fixed programs and are deployed widely, so these techniques have traditionally been used for finding vulnerabilities in this domain. While generating input data and taking the measurements is automated, energy trace analysis is mostly manual.

While the measurements and methodology presented in this section do not directly aim at comprehending a program (in fact, the attacked program must be understood very well to facilitate such an attack), these attacks bear some similarity to our goals. The attacks presented here know the energy behavior of different code segments and use this knowledge to identify which segments are executed in which order. We think that similar approaches can be used the other way around: if the execution order is known, the energy behavior of code segments can be identified. Also, similar to the counter-measures like clock jitter, when dealing with general-purpose programs on general-purpose computers, features like operating system task scheduling or garbage collection of managed languages distort the measurements. The strategies for dealing with such distortions developed in the presented attacks could be helpful to overcome problems, which are to date usually identified and excluded from the analysis by hand (cf. [10])

2.3 Best Practices

Gottschalk et al. [8, 9] collect several so-called energy code smells, which indicate a bad energy utilization, and propose refactorings to improve the energy behavior. The smells and refactorings they describe are limited to the domain of Android smart phone apps. The effectiveness of their refactorings is evaluated by measuring the energy consumption before and after refactoring. For this purpose, they make long-term energy measurements and compare the accumulated energy consumption after two hours. For the measurements, they use the default Android BatteryManager API and vendor-specific interfaces. Energy code smells used by Gottschalk et al. are, e.g., related to loading remote content [1] or binding of hardware components [15].

Pathak et al. [16] investigate a specific scenario leading to unnecessary energy drain in smart phones, namely accidentally keeping the smart phone awake. They describe three so-called no-sleep bugs. The first one, "no-sleep code path" is to acquire a wake lock without releasing it again. A related bug is called "no-sleep race condition"; here a race condition in a multi-threaded app non-deterministically hinders the app from releasing a wake lock. Lastly, "no-sleep dilation" bugs occur when potentially long-running operations are performed between the acquisition and the release of a wake lock. The energy consumption is not measured directly, but for several apps, the time is compared that it takes to drain the battery.

Vetro et al. [22] describe energy code smells for general-purpose programs. For this purpose, they investigate the code smells detected by code checkers for C++ and Java code, namely Cpp-Check and Findbugs. For each smell that the authors consider potentially relevant for energy efficiency, they wrote a pair of functionally equivalent code segments, one with and the other without the smell. These code segments were executed on a Waspmote V1.1 board with an ATmega 1281 microcontroller with a CPU frequency of 8 MHz. For each run, the power consumption was measured, and they identified five code smells which hint at inefficient energy behavior: DeadLocalStore, NonShortCircuit, ParameterByValue, RepeatedConditionals, SelfAssignment.

3 Conclusion

For this survey we set out to find development tools that support software developers in optimizing or debugging the energy behavior of their code. While the literature and tool search was not performed in a very systematic way, we nevertheless are confident that we gained a representative overview of this field of research. The sobering result of this survey was that there are hardly any actual ready-to-use development tools; those that we found are all platform-specific. There is also a promising result of our survey, namely there are a lot of approaches for assessing the energy consumption of software systems, which lets us hope that such approaches (or advancements thereof) can be leveraged for building required development tools.

We discussed eleven approaches for measuring the energy consumption of software systems and compared them by the supported architecture, measurement granularity and assumptions they make on the measurement environment. About half of the presented approaches target general-purpose, half special-purpose architectures. However, we can report that there is a peak of literature on the energy consumption in Android systems. Our survey has also shown that the granularity of energy measurements is in most cases relatively coarse, although there are indicators that a finer granularity leads to more accurate results. For completeness, we also investigated best practices for energy efficient code. While these approaches do not measure the actual consumption but only search for predefined patterns in code that hint at bad energy usage in code, they can still provide feedback to developers and thus reasonably meet our initial objective.

References

[1] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[2] X. Charvet and H. Pelletier. Improving the dpa attack using wavelet transform. In *NIST Physical Security Testing Workshop*, volume 46, 2005.

[3] M. Couto, J. Cunha, J. P. Fernandes, R. Pereira, and J. Saraiva. Greendroid: A tool for analysing power consumption in the android ecosystem. In *Scientific Conference on Informatics, 2015 IEEE 13th International*, pages 73–78. IEEE, 2015.

[4] G. e Sustainability Initiative et al. Smarter2030, ict solutions for 21st century challenges. *Belgium, GeSI, Accenture Strategy*, 2015.

[5] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan. Full-system power analysis and modeling for server environments. In *Workshop on Modeling Benchmarking and Simulation (MOBS) at ISCA*. International Symposium on Computer Architecture-IEEE, 2006.

[6] K. Eder, J. P. Gallagher, P. López-García, H. Muller, Z. Banković, K. Georgiou, R. Haemmerlé, M. V. Hermenegildo, B. Kafle, S. Kerrison, et al. Entra: Whole-systems energy transparency. *Microprocessors and Microsystems*, 47:278–286, 2016.

[7] M. A. Ferreira, E. Hoekstra, B. Merkus, B. Visser, and J. Visser. Seflab: A lab for measuring software energy footprints. In *Green and Sustainable Software (GREENS), 2013 2nd International Workshop on*, pages 30–37. IEEE, 2013.

[8] M. Gottschalk, J. Jelschen, and A. Winter. Saving energy on mobile devices by refactoring. In *28th International Conference on Informatics for Environmental Protection: ICT for Energy Effieciency*, pages 437–444. BIS-Verlag, 2014.

[9] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter. Removing energy code smells with reengineering services. In *Informatik 2012, 42. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, volume 208 of *LNI*, pages 441–455. GI, 2012.

[10] S. Hao, D. Li, W. G. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 92–101. IEEE, 2013.

[11] A. Hindle. Green mining: A methodology of relating software change to power consumption. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 78–87. IEEE Press, 2012.

[12] Intel Corporation. Enhanced intel speedstep technology for the intel pentium m processor, 2004.

[13] X. Li and J. P. Gallagher. A top-to-bottom view: Energy analysis for mobile application source code. *CoRR, abs/1510.04165*, 2015.

[14] H. Malik, P. Zhao, and M. Godfrey. Going green: An exploratory analysis of Energy-Related questions. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 418–421, May 2015.

[15] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[16] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[17] G. Pinto, F. Castor, and Y. Liu. Mining questions about software energy consumption. In *11th Working Conference on Mining Software Repositories*. ACM New York, NY, USA, 2014.

[18] C. Seo, S. Malek, and N. Medvidovic. Estimating the energy consumption in pervasive java-based systems. In *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, pages 243–247. IEEE, 2008.

[19] Silicon Labs. Energy debugging tools for embedded applications.

[20] V. Tiwari, S. Malik, A. Wolfe, and M.-C. Lee. Instruction level power analysis and optimization of software. In *VLSI Design, 1996. Proceedings., Ninth International Conference on*, pages 326–328. IEEE, 1996.

[21] J. G. van Woudenberg, M. F. Witteman, and B. Bakker. Improving differential power analysis by elastic alignment. In *Cryptographers' Track at the RSA Conference*, pages 104–119. Springer, 2011.

[22] A. Vetro, L. Ardito, G. Procaccianti, and M. Morisio. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In *Proceedings of the Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies(Energy)*, pages 34—-39, 2013.

[23] C. Wilke, S. Götz, and S. Richly. Jouleunit: a generic framework for software energy profiling and testing. In *Proceedings of the 2013 workshop on Green in/by software engineering*, pages 9–14. ACM, 2013.

[24] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 105–114. IEEE, 2010.