Code-level Energy Hotspot Localization via Naive Spectrum Based Testing

Roberto Verdecchia, Achim Guldner, Yannick Becker, and Eva Kern

Abstract With the growing adoption of ICT solutions, developing energy efficient software becomes increasingly important. Current methods aimed at analyzing energy demanding portions of code, referred to as *energy hotspots*, often require *ad-hoc* analyses that constitute an additional process in the development life cycle. This leads to the scarce adoption of such methods in practice, leaving an open gap between source code energy optimization research and its concrete application. Thus, our underlying goal is to provide developers with a technique that enables them to efficiently gather source code energy consumption information without requiring excessive time overhead and resources. In this research we present a naive spectrum-based fault localization technique aimed to efficiently locate energy hotspots. More specifically, our research aims to understand the viability of spectrum based energy hotspot localization and the tradeoffs which can be made between performance and precision for such techniques. Our naive yet effective approach takes as input an application and its test suite, and utilizes a simple algorithm to localize portions of code which are potentially energy-greedy. This is achieved by combining test case coverage information with runtime energy consumption measurements. The viability of the approach is assessed through an empirical experiment. We conclude that the *naive spectrum based energy hotspot localization* approach can effectively support developers by efficiently providing insights of the energy consumption of software at source code level. Since we use processes already in place in most companies and adopt straightforward data analysis processes, naive spectrum based energy hotspot localization can reduce the effort and time required for assessing energy consumption of software and thus make including the energy consumption in the development process viable. As future work we plan to (i) further investigate the tradeoffs between performance and precision of spectrum based energy hotspot approaches (ii) compare our approach to similar ones through large-scale experiments. Our ultimate goal is to conceive *ad-hoc* tradeoff tuning of performance and precision according to development and organizational needs.

Roberto Verdecchia
Gran Sasso Science Institute, L'Aquila, Italy
Vrije Universiteit Amsterdam, Amsterdam, the Netherlands. e-mail: roberto.verdecchia@gssi.it

Achim Guldner, Yannick Becker
University of Applied Sciences Trier, Environmental Campus Birkenfeld, Birkenfeld, Germany.
e-mail: a.guldner@umwelt-campus.de, y.becker@umwelt-campus.de

Eva Kern
Environmental Campus Birkenfeld, Birkenfeld
Leuphana University Lueneburg, Germany. e-mail: mail@nachhaltige-medien.de

1 Introduction

Nowadays, software systems are becoming more and more ubiquitous. With the ever increasing adoption of software solutions, the energy consumption of the underlying hardware on which the software is run is no longer negligible. This becomes particularly evident if data centers are considered. In fact, an increasing amount of resources is allocated nowadays to ICT [37], and the overall consumption of data centers alone accounted between 1.1 % to 1.5 % of global energy consumption already back in 2010 [22]. In addition to the environmental impact of ICT, another aspect deeply influences the software energy efficiency research field, namely mobile devices. Mobile devices are notably characterized by a limited amount of resources at their disposal. Hence, energy efficiency is becoming an increasingly crucial performance and usability concern for mobile users. During the years, numerous research efforts focused on the evaluation of the energy efficiency of software products [38]. Such research results to be homogeneous w.r.t. the domain and approach utilized in order to assess the energy consumption of software systems. In particular, due to the recent popular shift towards the cloud paradigm, an increasing number of studies focus on possible software optimizations aimed to improve the energy efficiency of large data centers [16, 15, 7]. Following the same rationale of popularity, a wide range of approaches aimed at assessing, measuring and optimizing energy consumption of mobile applications have been reported in the literature [11, 27, 10].

Energy efficiency assessment analyses can be divided into two macro-categories, namely *static* and *dynamic* analyses, according to the specific technique adopted to evaluate the energy efficiency. As the name suggests, static analyses commonly evaluate the energy efficiency of source code without compiling and executing it. Instead, such techniques rely on analyzing the source code in order to map computations to CPU cycles or energy models, in order to obtain an estimate of the energy consumed at runtime [36]. Static analyses are usually characterized by being technology dependent, i.e. specifically tailored for a programming language or framework. In addition, while the time required to execute static analyses is generally low, the output often results to be an approximation of the energy consumed, lacking empirical evidence of the calculated outcomes. In contrast, dynamic analyses are usually carried out by directly measuring the run-time energy consumption of a software application. Such approaches are usually adopted to validate hypotheses through empirical experiments or carry out ad-hoc energy efficiency assessments.

Dynamic analyses, in contrast to the static ones, provide precise information on the energy that is consumed by the hardware on which the analyzed software application is executed, and hence result to be generally more reliable in terms of precision w.r.t. the static ones. Nevertheless, dynamic analyses require more time to be carried out than static analyses, as the time required to execute some functionality is needed in order to gather measurements. In addition, dynamic analyses are often characterized by time-demanding set-up times, which are necessary to identify the variables to be considered and to implement the experiment framework in which the experiment is carried out. In addition, dynamic analyses heavily rely on the identification of a significant set of use case scenarios through which the soft-

ware application under analysis has to be load tested. Such a process is crucial, as, in case of an ill-suited selection of use case scenarios, the dynamic analysis might lead to inconclusive or insignificant results. Due to its importance, the selection, implementation, and execution of use case scenarios often results in a time demanding process. It also requires knowledge of the software under analysis and resources to be allocated for such a task. As a primary or secondary end result, static and dynamic analyses often lead to the identification of *energy hotspots*, i.e. scenarios in which executing an application causes the underlying hardware on which the application is run to consume abnormally high amounts of energy [5]. In general, energy hotspot localization often results in time- and resource-consuming processes, leading to their scarce adoption in industrial settings.

In this paper, we adapt the knowledge of spectrum-based fault localization research in order to evaluate if similar techniques can be utilized in order to detect portions of source code which consume an anomalous amount of energy. Spectrum-based fault localization techniques rely on the coverage information of test cases in order to pinpoint portions of source code which are more likely to contain faults. We investigate if a *naive* spectrum based testing technique can be used to efficiently localize energy hotspots. As a result, we lie the groundwork towards a better understanding of potential tradeoffs which lie between performance and precision of spectrum based energy hotspot localization techniques.

The remainder of the paper is structured as follows: in Section 2 the literature related to our study, i.e. considering software energy efficiency assessment, is presented. The naive approach combining program spectra and empirical energy measurements that we conceived for energy hotspot localization is presented in Section 3. In Section 4 we document the details and results of the empirical experiment we devised in order to assess the viability of our approach. In Section 5 the threats to validity of our experiment are reported. In Section 6 we discuss the benefits and drawbacks of our approach and the results of the experiment. Finally, in Section 7 we report the conclusions and the future steps this research leads to.

2 Related Work

Energy efficiency measurements raise the interest in different fields, e.g. data centers, mobile devices, and cloud computing. Hence, it is not surprising that there are plenty of research activities addressing methods how to measure the energy consumption of ICT, even if the focus on the software side is quite new in this context [30, 38]. Thus, the following section will provide an insight into the research field of green software engineering, focusing on energy efficiency of software, as well as corresponding measurement methods. We do not claim to present a comprehensive literature analysis, as many literature studies and extensive literature discussions in this context have been published, e.g. by Procaccianti et al. [33], Bozzelli et al. [8], Zein et al. [41], and, focusing on mobile applications, Vásquez et al. [24]. According to Calero et al. [9], compared to other sustain-

ability issues of software, energy, or rather power consumption is much more addressed in context of measurements. Besides differentiating between static and dynamic energy efficiency assessment analyses (see Section 1), the approaches available in the literature can be distinguished according to the methods used to carry out the measurement (black-box/white-box measurements, see e.g. [20, 4]), the system under test (mobile-device/desktop-computer/server-system), and the strategy for the acquisition of data (energy consumption measurement/estimation).

Black box measurements regard the system under test (SUT) as a black box to be tested without any knowledge of the implementation of the software [4]. In contrast, white box measurements analyze the software product based on its source code. One example for this latter type of analysis is the energy efficiency assessment of different programming languages and algorithms to solve a specific problem, e.g. the work of presented by Rashid [34] considering sorting algorithms. In a study by Johann et al., source code instrumentation is used to locate resource intensive parts of programs in order to improve them [20]. This is also done by Procaccianti et al. [33] when testing energy efficient software practices and by Verdecchia et al. [39] in the context of energy optimizations through code-smell refactoring. Most of such methods are based on predefined Use Case Scenarios (UCS), through which software applications are load-tested.

One big issue in the context of energy consumption of software lies in the field of mobile devices. Similarly to software developers, users are not aware of the energy consumption of mobile software applications [28]. Nevertheless resource-related complaints have a high relevance in the context of mobile applications [21]. Thus, researchers conceived numerous approaches aimed to assess the energy consumption in context of mobile devices: In order to automatically test mobile applications, Linares-Vásquez et al. [24] present a conceptual framework following the CEL principles: Continuous, Evolutionary, and Large-scale. Wilke et al. [40] provide a prototypical realization of energy profiling as a service for mobile applications. This service is conceived to support research groups and application developers in order to reduce the effort to build up a dedicated testing environment. Ahmad et al. [3] present a comprehensive comparison of different energy profiling methodologies in case of mobile applications: they compare different energy profiling schemes, hardware and software based profiling methods, and introduce a thematic taxonomy in this context. In the context of desktop computers and server systems, end user does not directly recognize the energy consumption in the form of battery life. Nevertheless, we also consider these two fields as relevant in case of finding solutions for energy efficient software.

The concept of efficiently identifying energy consuming software components is gaining more and more attention in the literature. In the context of mobile applications, Pathak et al. [29] present an approach aimed at assessing the energy consumption of four different program entities, namely processes, threads, subroutines, and system calls. In comparison, our approach results to be language independent, and focuses on distinct entities, namely program branches, functions, and lines of code. In a related work by Liu et al. [25] the energy inefficient usage of mobile sensors and related data is assessed by simulating the runtime behavior of an application.

Apart from the context considered (which in our case is not focusing exclusively on sensor usage and the mobile ecosystem) our study differs by being driven by real measurements, i.e. without simulating software energy consumption through real time hardware measurements. Li et al. [23] present a tool based on path profiling to correlate energy measurements to single lines of code. In contrast to this study, our approach considers three distinct levels of granularity, namely program lines, functions and branches. Additionally, our study has a different underlying goal, namely exploring the viability that naive approaches can have to detect energy hotspots at source code level. Investigating such hypothesis lies the groundwork towards a better understanding of the tradeoffs between precision and performance of spectrum based energy hotspot localization techniques, striving towards conceiving simple yet effective analyses. In the work of Hindle et al. [17] a framework to automatically gather energy consumption data of mobile applications at test case level is presented. Our work builds on such a concept by adopting a refined level of granularity. This is done in our approach by computing which portions of code covered by the test cases are more/less energy efficient.

The research which relate the closest to our study are the ones of Pereira et al. [31, 32]. In their studies, a tool under development named SPELL is presented. The tool makes use of an adaptation of spectrum-based fault localization [2] by considering three different parameters for each test case, namely energy consumption, execution time, and number of executions. In comparison, our approach exclusively relies on the variation of energy of each test case execution. Additionally, our approach does not rely on the coefficient of test case similarity, but instead evenly distributes the energy consumption among the *items* covered by the test cases. While the goal of the SPELL tool [31, 32] lies in providing the means to precisely locate energy hotspots in source code, our study aims to investigate if more naive approaches can be used to effectively locate them. This enables us to lie the groundwork to accurately understand through empirical experimentation the magnitude of the tradeoff that lies between performance and precision of spectrum based energy hotspot localization. Through this initial step we aim to provide the basis in order to experiment with incremental depth the impact of performance w.r.t. precision. This would potentially enable a fine grained tuning of the parameters, in order to adjust them according to development and organizational needs.

Summarizing, research activities focusing on subjects related to the energy estimation of software are carried out. The approach presented in this paper aims to investigate more in depth a specific aspect of a particular technique, namely spectrum based energy hotspot localization. More specifically, we carry out a preliminary investigation in order to understand if a naive spectrum based approach, which is potentially more performant but also less precise than existing techniques, can be utilized to effectively localize energy hotspots in source code. This enables us to lay the groundwork towards understanding such tradeoffs by empirical means, with the final goal of considering and tuning the approach on an *ad-hoc* basis.

3 Approach

As presented in Section 2, a commonly adopted technique used to empirically evaluate the energy efficiency of software applications is to utilize UCSs in order to load-test an application and measure the energy consumption of the underlying SUT. Nevertheless such a technique binds the energy consumption exclusively to UCSs, which might vary in number of steps, complexity and required execution time. Hence, it is hard to remap the energy consumption induced by the UCS execution to a particular portion of code, as the only data available is the atomic energy consumption, relative to the execution of entire UCSs.

In this paper we present a naive approach which combines *program spectra* information [35] (also referred to as code coverage) with runtime hardware energy consumption in order to identify which portions of code are potentially energy-greedy. The essential intuition behind our approach is to take advantage of commonly pre-existing artifacts of software applications, namely test suites and test case coverage data, and combine this information with runtime energy measurements in order to identify energy-greedy portions of code. Specifically, the approach is composed of two phases, the first of which consist of two independent steps that can be executed in arbitrary order. An overview of the phases and steps of the approach are presented in Figure 1.

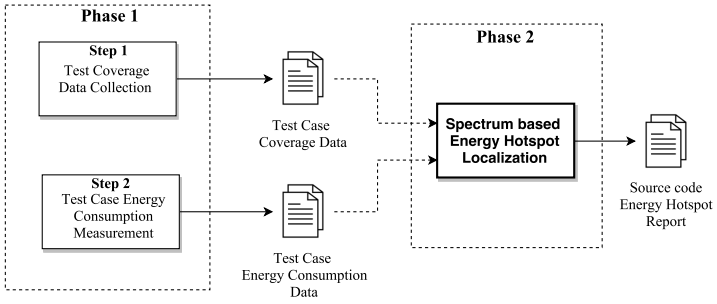

Fig. 1 Steps and phases of our approach

More specifically, the phases of our approach consist of:

Phase 1 - Step 1: Test case coverage data collection. The goal of this step is to gather the coverage information of each available test case. The coverage criteria considered (e.g. function, branch, or line coverage) will result in the granularity at which the energy hotspot localization will be carried out. If coverage information is already available, e.g. if regression testing processes are in place, this step can be skipped in order to accelerate the process.

Phase 1 - Step 2: Test case energy consumption measurement. The goal of this step is to measure the energy required by the SUT in order to execute each test case individually. This step consists of (i) the individual execution of each available test case and (ii) the measurement of the energy consumption of the SUT on which

the application is running. In order to gather enough statistical data, each test case can be executed multiple times (as further detailed in Section 4.2). In order to carry out this step, a power meter (PM) with the ability to export the measurement data is required to monitor the power consumption of the SUT.

Phase 2: Naive spectrum based energy hotspot localization. The third step of the approach consists of the combination of the information gathered from Step 1 and 2. In particular, in order to localize potential energy hotspots, the approximate energy consumption of each ***item*** of the coverage level considered (e.g. function, branch or source code line) is calculated by (i) iterating over the coverage information of each test case to count how often each item is covered, and (ii) assigning to each item its average energy consumption.

Intuitively, we want to identify those items that are more often involved in energy-hungry calls during program execution. This means that for each covered item i, the corresponding energy E_i is calculated as

$$E_i = \frac{1}{n_i} \sum_{j=0}^{n_i} \frac{\overline{E_j}}{k_j} \tag{1}$$

where n_i is the total number of test cases in which item i is covered; for each test case j, $\overline{E_j}$ is its average energy consumption and k_j is the number of items it covers. In this way, the energy $\overline{E_j}$ measured for each test case j is equally distributed among the items k_j covered by the test case. Subsequently, the estimated energy E_i of each item i is averaged among all n_i energy measurements $\frac{E_j}{k_j}$ of each test case j covering the item, identifying which items are more often involved in energy-hungry calls.

4 Experimental proof of concept

In order to confirm the viability of the approach, we devised an experimental measurement setup and analyzed the results we gathered from it. To carry out the experiment, we selected an artifact available in the **S**oftware-artifact **I**nfrastructure **R**epository (SIR)[1] [13]. We adopted this repository as it contains a set of software artifacts with already implemented test suites and relative coverage data. With the artifacts, we devised a controlled measurement experiment, where we assessed the energy consumption of a system under test (SUT) running the software, following ISO/IEC 14756 as introduced by Dirlewanger [12]. More specifically, we collected runtime energy measurements, and subsequently distributed the energy consumption over the test cases and, using the coverage information of the test cases, over the functions, branches and source code lines, following the approach described in Section 3. From SIR, we selected the Unix command-line utility program "`grep`". The measurement setup, data collection process, and results are described below. A

[1] The repository is available online at http://sir.unl.edu [Retrieved 2018-01-05]

replication containing the software artifacts and scripts utilized to run the experiment, process the data, and analyze the results is available online[2].

4.1 Measurement setup

As described, we adopted a measurement setup for our empirical measurements, following ISO/IEC 14756. To measure the energy consumption of a software product (or in our case a software artifact), the standard suggests to let a computer system (in our case a desktop computer[3]) execute a defined set of tasks and monitor the consumption of the system under test at the hardware level. Figure 2 depicts the measurement setup utilized to record the energy consumption of a SUT that is induced by a software product. Before the software product is installed on the SUT, all possible background processes, such as automatic updates, virus scanners, indexing- and backup processes, are deactivated and a baseline measurement is conducted. This is done to ensure minimal side effects. The workload generator then triggers the execution of the test case on the SUT (in our case by means of a Bash script). The power supply of the SUT is monitored by a power meter[4], which collects the data. The data is aggregated in a centralized data storage and then analyzed. The measurement data and workload statistics are synchronized by means of time stamps.

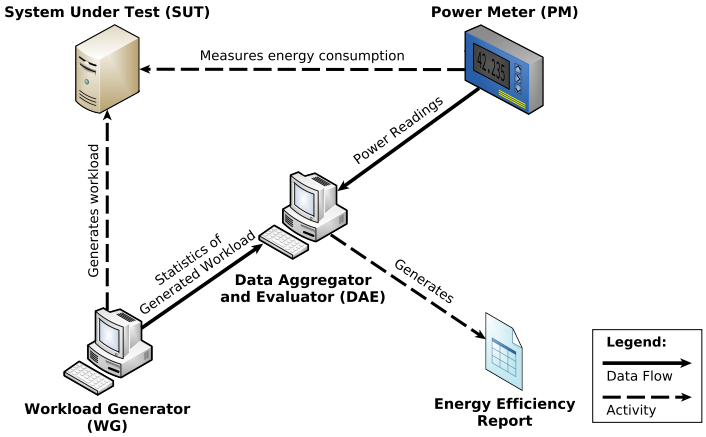

Fig. 2 Setup for measuring the energy consumption of software

[2] https://github.com/energyHotspots/EnviroInfo2018/

[3] Hardware specifics of the SUT adopted for experimentation: AMD Ryzen 7 1700, GeForce GTX 1060, 16 GB DDR4 RAM @ 2133 MHz, MSI B350 PC Mate, running Ubuntu 16.4 LTS.

[4] Power meter utilized for the measurements: Janitza UMG 604 Power Analyser. Sampling rate: 20 kHz, resolution: 10 mV, 1.0 mA.

The test script repeats the execution of each use case scenario (in our case 16 times[5]). For each measurement, we record the power input, which is directly averaged per second by the power meter. We store the measurement data together with the log data from the load generator in a database, in order to be able to accurately map test cases to their energy consumption. Details on the measurement procedure can also be found in [14]. In the following, we present the results of the measurements from the case study.

4.2 Data collection

The aim of the work described in this section was to create a proof of concept for the approach we conceived. In order to do so, we selected a software artifact available on SIR namely, "`grep`" version 3. The test suite available for this application consists of 808 test cases. `grep` is a linux command-line utility, that processes text inputs and prints lines which match a specified expression. The SIR test cases consist of `grep` calls with parameters of the sort:

```
grep -e if -e else ../inputs/grep1.dat
```

In this exemplary case, the parameter `-e` is the expression that is searched for in the input file `grep1.dat`. After several trial runs, we chose to repeat each test case 5000 times for one test run. This method stretches the execution time of the test cases to an average of 11.38 seconds to ensure viable measurements with our power meter. Additionally, as we want to measure exclusively the energy consumption induced by `grep`, we write all results of the calls to the null device in order to avoid measurement noise caused by the operations required by the operating system to print the text.

After all data was collected, we analyzed the power measurement by first averaging over the 16 times 5000 measurements of each test case. Figures 4 and 3 show the mean power consumption values of each call (i.e. test case) and the power consumed, plotted against the average duration of that call, respectively.

To evaluate the precision of the measurement setup we calculated the average standard deviation of the power measurement per test case to be $3.23 \cdot 10^{-3}$ milliwatts (with a mean value of 0.797 milliwatts) and the average standard deviation for the duration to 36.5 milliseconds (with a mean value of 11.38 seconds). To factor in the duration of the individual calls, from this point forward, we use the average energy per test case, which is calculated as the average power consumption, multiplied with the average duration (this results in the unit millijoule [mJ]).

Finally, we take into consideration the test suite coverage information available on the SIR repository for the artifact considered. The coverage information is separated into the three categories

[5] One use case scenario consists of all test cases from the SIR test suite for the program "grep". Each test case was repeated 5000 times in each scenario. Thus, each test case was run 80000 times (see also Section 4.2).

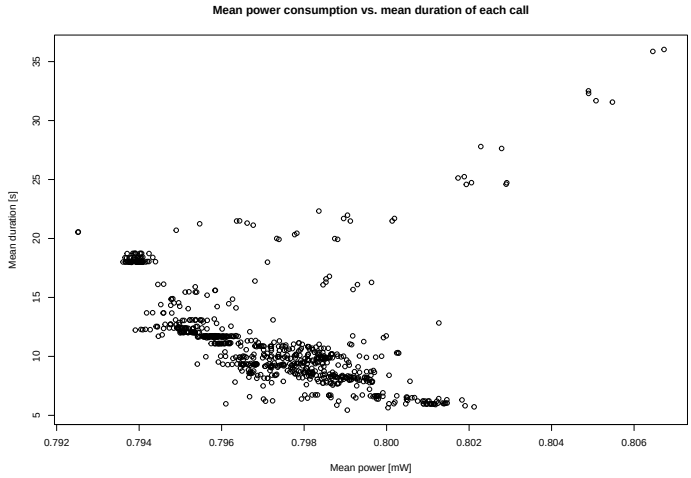

Fig. 3 Mean power consumption in milliwatts, plotted against the mean duration of the test case in seconds

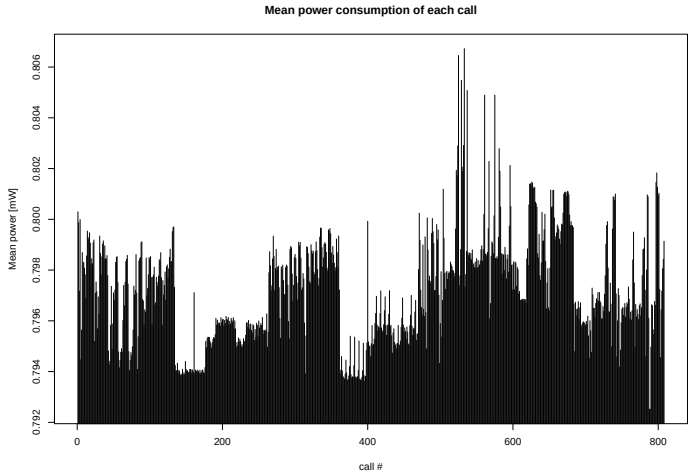

Fig. 4 Mean power per test case in milliwatts

- function coverage (107 functions),
- branch coverage (1801 branches), and
- line coverage (13372 lines of code, including comments).

To take the final step towards assessing the average energy consumption on a function-, branch-, and line-based level, we simply iterate over the coverage information of each test case for the three categories, count how often each item is cov-

ered, and assign it its average energy consumption. Specifically, energy consumption of each item *i* is calculated by applying Formula 1 reported in Section 3 (Step 2).

4.3 Results

In this section we report the results gathered for our experimental proof of concept. More specifically, in order to carry out a preliminary evaluation of the viability of our approach, the results gathered through the reported experiment were manually inspected. This operation consisted in (i) examining the processed data, (ii) selecting a subsample of items according to which ones resulted to be potentially more/less energy-greedy (iii) inspecting the source code of the selected items to get more insights of the results. In order to be able to carry out an in-depth manual evaluation of the gathered results, we concentrated our efforts in inspecting the items identified at function level. In order to do so, we considered the calculated average energy consumption estimates for each function covered by the test cases. Functions that resulted not to be covered by the test suite were not considered during the hotspot identification, as no coverage data was available as input for our approach.

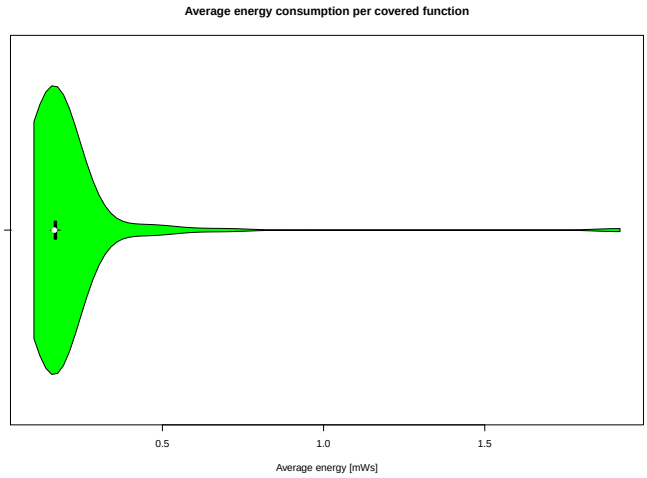

Fig. 5 Distribution of the average energy consumption of covered functions

Figure 5 presents the measurements of the average estimated energy consumption per function. We can observe clearly that there is one function with a particularly high energy reading. For Figure 6 we additionally assigned a color palette to the energy consumption to ease through graphical means the identification of the most energy consuming functions. In our case we can observe that function `usage()` consumes much more energy than all others. This visual example shows the po-

tential of the approach to rapidly and efficiently report the results of the hotspot analysis. A similar visualization technique could be used in a future tool (e.g. an IDE plugin) to directly point the developer to the energy hotspots in their source code editor.

4.4 Approach viability

From the manual inspection of the results gathered from the experiment, we can conclude that our naive approach can successfully detect energy-greedy items. As the manual inspection of source code resulted in a time consuming process, we concentrated out effort in the in-depth analysis of a subset of prominent items identified at the function coverage level. Specifically we manually inspected the source-code of the 6 most energy-greedy functions and 11 of the least energy-consuming ones in order to assess the viability of our approach (see Figure 6). In order to comprehensively interpret the data, in addition to the hotspot analysis results, we also inspected the unprocessed test case coverage information included in the replication package. In the reminder of this section we briefly discuss the manual inspection results.

Regarding the most energy-consuming functions, 6 were identified, namely: `usage()`, `fatal()`, `ck_atoi()`, `Fcompile()`, and `Fexecute()`. From the manual investigation of the source code of the function `usage()`, two main reasons were identified for its high energy consumption, namely: (i) the function is always part of test cases covering only few functions, i.e. the energy overhead required to start and end the `grep` process highly influences the average energy consumption, and (ii) the function makes use of output printing more extensively than all other functions. From this occurrence, we can conclude that the energy hotspot is caused by the expected behaviour of the function, and hence does not require refactoring.

From the manual inspection, the high energy readings of `fatal()` appear to be caused, as for the function `usage()`, by the appearance of such functions in "short" test cases. The energy hotspot is therefore caused by the application start and end energy overhead, and should hence not be considered for refactoring processes.

The `xrealloc()` and `ck_atoi()` functions resulted to be energy-greedy, presumably, due to the involved computations, namely: expensive memory operations and iterating, checking and converting each character of a string into numeral. A careful inspection of such occurrences should be carried out by the application developers in order to assess if such operations can be optimized.

Through the inspection of the implementation of `FCompile()` and `Fexecute()`, we concluded that their high-energy consumption is caused by complex computations, involving nested loops and `goto` statements. As for the previous example, further inspection by the developers of the application should be conducted to investigate possible energy optimizations of this hotspot.

Regarding the least energy consuming functions, as can be seen in Figure 6, these resulted in most of the cases to be simple character comparison functions, such as `is_alnum` (is alphanumeric), `is_digit` (is numeric), `is_lower` (is lower

case), etc. This leads us to conclude that simple atomic functions, which intuitively consume little energy, are also correctly identified by our approach.

In summary, from the manual inspection of the results, we can conclude that our approach is able to detect both low- and high-energy-consuming portions of code.

Fig. 6 Energy consumption per function

5 Threats to Validity

In this section we report the threats to validity of the experiment adopted for the preliminary assessment of the viability of our approach. It is important to bear in mind that this research focuses on the exploration of the possibilities that lie in adapting a naive spectrum-based fault localization technique in order to identify energy hotspots at source code level. Therefore, the focus of the research lies in the presentation of the potential of such concepts, rather than on their extensive evaluation by empirical means. Nevertheless, in order to provide an overview of the main shortcomings of the preliminary empirical evaluation reported in this paper, the major threats to validity that have to be considered for this research are presented below.

External Validity. The adoption of a single software artifact for the experiment results to be our major threat to external validity. One of the main benefits of our approach is that it is language independent, i.e. the only requirements necessary to carry out the hotspot analysis are the availability of (i) a test suite with coverage

information, and (ii) a power meter. In order to mitigate potential threats to external validity, as future work, we plan to extensively evaluate the approach on a large set of applications, ranging different languages and technologies.

Internal Validity. In order to mitigate potential threats to internal validity, we took precautions to minimize side effects in the measurements, performed baseline measurements of the SUT energy consumption (see Section 4.1) and executed each test case multiple times (see Section 4.2) in order to ensure the statistical relevance of the data. Furthermore, in order to evaluate the precision of the measurement setup, we calculated the standard deviation of the power measurements and execution times.

Construct Validity. In order to ensure that a relevant and indicative software application and test suite was considered, we chose an artifact taken from a well established repository, which is widely used for scientific software testing research, namely SIR [13]. While constituting a representative measurand, the potential that our approach has to be language independent was not explored further, as we concentrated on the evaluation of an artifact implemented in the programming language *C*. This constitutes the most prominent threat to the construct validity of our empirical evaluation. As for the threats to construct validity, the strategy required to mitigate this threat consists of considering a large set of heterogeneous applications, which will be adopted for experimentation in the future.

Conclusion Validity. The most notable threat to conclusion validity is constituted by the analysis method adopted to inspect the results. In fact, in order to evaluate the results in depth, a manual evaluation was adopted. Nevertheless, this process is error-prone and sensitive to potential subjective interpretation of the results. Additionally, such process was carried out by researchers which did not develop the application, further increasing this threat to validity. In order to mitigate such threat, three researchers independently carried out the manual inspection of the results and jointly discussed them in order to identify potential occurrences of divergent interpretations. An additional threat to conclusion validity is the inspection of a subset of typology of items (namely a subset of high/low energy consuming functions), which was adopted due to time constrains, as the manual inspection of the entirety of all covered item types resulted to be a very time-consuming process.

In order to mitigate the identified threats to conclusion validity of the study, in our future work we envision to involve developers of the analyzed applications in the evaluation process.

6 Discussion

In this paper, we present the opportunities that lie in utilizing spectrum based fault detection techniques to localize potential energy hotspots at source code level. Adapting spectrum based fault detection techniques for energy hotspot localization can lead to many potential benefits, but entails also some potential drawbacks. In

this section we discuss the benefits and drawbacks we identified by considering the approach we conceived:

6.1 Approach Benefits:

(i) *time-efficiency*: the approach does not require much time to set-up and carry out, as it relies on artifacts which are commonly present in software environments (namely applications and test suites), and requires the time needed to collect coverage information and execute test cases[6]. The strength of our approach lies in the time required to calculate the average energy per item. In fact, compared to other similar approaches, our naive approach result to be less complex from a computational point of view. The gain in effectiveness might negatively impact the precision of the analysis. Nevertheless this latter point requires serious empirical experimentation, that is left as part of out future work.

(ii) *language independence*: as similar approaches, our conceived one is language independent and potentially technology agnostic. The only technology requirements are that the application to be analyzed is provided with a test suite and that coverage data is obtainable through profiling tools.

(iii) *test relevance reliability*: the approach takes advantage of pre-existing test suites of the application to be analyzed. The identification of the portions of code to be inspected is therefore delegated to the software testing processes. Hence, the depth of the analysis and the items considered is exactly the same as one utilized by developers to test the application. This means that our approach does not rely on *ad-hoc* identified UCSs, which usually characterize empirical energy measurement approaches.

(iv) *variable granularity*: our approach specifically takes into account different levels of granularity, according to the coverage criteria considered (e.g. function, branch, and line coverage). Therefore the energy hotspot localization can be tuned according to different levels of depth in order to better fit into time constraints, desired precision or other requirements.

(v) *driven by real measures*: the approach is based on the refinement of exact energy consumption measurements and not on simulated results based on source code inspection. This means that hardware specifications can be easily considered by executing the measurements on the hardware itself, and do not have to be simulated and approximated by theoretical means.

(vi) *outsourcing opportunities*: The presented approach requires a power meter. This gives the opportunity to software service providers to make available their measurement infrastructure in a Platform as a Service (PaaS) fashion in order to offer quick, reliable and easy to access empirical energy analyses of software products.

[6] Empirical measurements could be collected even during normal testing processes by measuring the runtime energy consumption during the test case execution. Additionally, if regression testing processes are adopted (i.e. coverage data is already available), coverage information does not need to be acquired, further accelerating the hotspot detection process.

This service could be integrated in existing IDEs, and requires exclusively a software application, its test suite, and optionally test coverage information.

6.2 Approach Drawbacks:

(i) *test suite dependence*: our approach does not entail a formal technique aimed to systematically explore code bases, but rather takes advantage of existing data collected from software testing processes which are already in place. For this reason our approach heavily relies on pre-made test suites: If a considered test suite is of low quality, so will potentially be the results of the energy hotspot localization analysis. This implies that, in order to produce accurate results, our approach requires test suites which are indicative of the real life usage of the analyzed application. Additionally, the input of the use cases should be representative of the common usage of the analyzed application. The less such assumption holds, the more the results of the hotspot analysis will diverge from the real world energy consumption of the application.

(ii) *power meter requirement*: the most prominent requirement of our approach is the necessity of a power meter in order to carry out the empirical measurements. While seldom available, these hardware components result to be rather inexpensive and are frequently deployed in academic and industrial environments which focus on hardware/software energy efficiency research. Furthermore, by considering the mobile device ecosystem, it is possible to take advantage of measurements gathered through one of the many energy profilers that exist to date [18], which have been effectively utilized in previous researches [26, 19]. Additionally, there are also approaches that aim at removing the need for power meters through software based estimation of power demand [6].

(iii) *localization uncertainties*: The most prominent drawback of our approach is its potential lack of precision. On the one side, our approach takes advantage of spectrum based testing techniques in order to efficiently localize potential energy hotspots. On the other side, it also inherits the intrinsic uncertainties that such techniques entail [1]. Additionally, our approach leverages precision in favour of performance. This is achieved by equally splitting the energy consumption of a test case among all its covered items. This can potentially lead to the calculation of fast estimates to the cost of a loss of precision. Our approach is conceived in order to better understand the potential tradeoffs that lie between performance and precision of spectrum based energy hotspot localization approaches. More specifically, if compared to related work, the approach presented in this study is potentially less precise but also less complex from a computational point of view. Nevertheless, from the proof of concept experiment provided in this study, we assessed that our approach results to be a viable option to effectively detect hotspots. As future work, we therefore plan to extensively compare our approach to other spectrum-based techniques, in order to understand to which extent the possible performance gains affected its precision. Additionally, we plan to improve the accuracy of our approach through

different techniques and assess how these affect efficiency and efficacy. More in detail, we plan to consider (i) the percentage of test cases covering each item, (ii) the variability of the empirical measurements in order to assess the precision of the estimates, and (iii) the adoption *ad-hoc* automated analyses to further inspect potential energy hotspots that are identified. This will lay the groundwork towards (semi-)automatic tuning of precision/performance of energy hotspot analyses based on the software development context and requirements.

7 Conclusion and Outlook

In this paper we evaluate the effectiveness of a naive spectrum-based fault localization technique for source code energy hotspot localization. More specifically, we conceived an approach which, taking as input an application and its test suite, efficiently localizes portions of code which are potentially energy-greedy based on straightforward data analysis processes. The viability of our approach was assessed through a preliminary empirical experiment that we devised.

This exploratory research presents the opportunity for many future steps, based on combination of empirical energy measurements and program spectra. More specifically, our approach is conceived in order to evaluate the tradeoffs that lie between efficiency and effectiveness of spectrum based energy hotspot localization techniques. The approach conceived for this study is only in its preliminary state, aimed to assess to which extent naive approaches can be adopted to locate energy hotspots. From the experiment result we assess that naive spectrum based analyses yield to promising results.

Additionally, we plan to incrementally extend the approach in order to increase its precision and evaluate potential performance losses. Among various techniques, we plan to consider (i) how many items cover the test cases, (ii) the variability of energy measurements, and (iii) more advanced spectrum based techniques which have been already conceived for fault localization [1]. In addition, we plan to carry out a comprehensive comparison of our naive spectrum-based approach with other hotspot localization approaches in order to better understand the tradeoffs between performance and precision that our approach entails. This will lay the groundwork to explicit the possible tradeoffs and potentially conceive *ad-hoc* tuning of parameters in order to set them according to development and organizational needs. In addition, we plan to assess the efficacy and effectiveness of the approach by considering for experimentation a large set of heterogeneous applications, ranging different languages and technologies.

References

1. R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.
2. R. Abreu, P. Zoeteweij, and A. J. Van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE Computer Society, 2009.
3. R. W. Ahmad, A. Gani, S. H. A. Hamid, F. Xia, and M. Shiraz. A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues. *Journal of Network and Computer Applications*, 58:42–59, 2015.
4. A. Banerjee, S. Chattopadhyay, and A. Roychoudhury. Chapter three-on testing embedded software. *Advances in Computers*, 101:121–153, 2016.
5. A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 588–598. ACM, 2014.
6. Y. Becker and S. Naumann. Software based estimation of software induced energy dissipation with powerstat. In *From Science to Society: The Bridge provided by Environmental Informatics*, pages 69–73. Shaker Verlag, 2017.
7. A. Beloglazov, J. Abawajy, and R. Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5):755–768, 2012.
8. P. Bozzelli, Q. Gu, and P. Lago. A systematic literature review on green software metrics. *VU University, Amsterdam*, 2013.
9. C. Calero, M. F. Bertoa, and M. Á. Moraga. A systematic literature review for software sustainability measures. In *Proceedings of the 2nd International Workshop on Green and Sustainable Software*, pages 46–53. IEEE Press, 2013.
10. Y.-F. Chung, C.-Y. Lin, and C.-T. King. Aneprof: Energy profiling for android java virtual machine and applications. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 372–379. IEEE, 2011.
11. S. K. Datta, C. Bonnet, and N. Nikaein. Android power management: Current and future trends. In *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*, pages 48–53. IEEE, 2012.
12. W. Dirlewanger. *Measurement and Rating of Computer Systems Performance and of Software Efficiency*. Kassel University Press, kassel, 2006.
13. H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.
14. A. Guldner, M. Garling, M. Morgen, S. Naumann, E. Kern, and L. M. H. Hilty. Energy consumption and hardware utilization of standard software: Methods and measurements for software sustainability. In *From Science to Society: New Trends in Environmental Informatics.*, pages 251–261. Springer International Publishing, 2017.
15. A. Hammadi and L. Mhamdi. A survey on architectures and energy efficiency in data center networks. *Computer Communications*, 40:1–21, 2014.
16. S. Harizopoulos, M. Shah, J. Meza, and P. Ranganathan. Energy efficiency: The new holy grail of data management systems research. *arXiv preprint arXiv:0909.1784*, 2009.
17. A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 12–21. ACM, 2014.
18. M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma. Modeling, profiling, and debugging the energy consumption of mobile devices. *ACM Computing Surveys (CSUR)*, 48(3):39, 2016.
19. R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. Ecodroid: An approach for energy-based ranking of android apps. In *Proceedings of the Fourth International Workshop on Green and Sustainable Software*, pages 8–14. IEEE Press, 2015.

20. T. Johann, M. Dick, S. Naumann, and E. Kern. How to measure energy-efficiency of software: Metrics and measurement results. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 51–54. IEEE Press, 2012.
21. H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.
22. J. Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, 9, 2011.
23. D. Li, S. Hao, W. G. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89. ACM, 2013.
24. M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 399–410. IEEE, 2017.
25. Y. Liu, C. Xu, and S.-C. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, pages 2–10. IEEE, 2013.
26. I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirovic. Assessing the impact of service workers on the energy efficiency of progressive web apps. In *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '17, Buenos Aires, Argentina, May, 2017*, pages 35–45, 2017.
27. A. W. Min, R. Wang, J. Tsai, M. A. Ergin, and T.-Y. C. Tai. Improving energy efficiency for mobile platforms by exploiting low-power sleep states. In *Proceedings of the 9th conference on Computing Frontiers*, pages 133–142. ACM, 2012.
28. C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2016.
29. 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 *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42. ACM, 2012.
30. B. Penzenstadler, A. Raturi, D. Richardson, C. Calero, H. Femmer, and X. Franch. Systematic mapping study on software engineering for sustainability (se4s). In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 14. ACM, 2014.
31. R. Pereira. Locating energy hotspots in source code. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 88–90. IEEE Press, 2017.
32. R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. Helping programmers improve the energy efficiency of source code. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 238–240. IEEE, 2017.
33. G. Procaccianti, H. Fernandez, and P. Lago. Empirical evaluation of two best practices for energy-efficient software development. *Journal of Systems and Software*, 117:185–198, 2016.
34. M. Rashid, L. Ardito, and M. Torchiano. Energy consumption analysis of algorithms implementations. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–4. IEEE, 2015.
35. T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Software Engineering-Esec/Fse'97*, pages 432–449. Springer, 1997.
36. 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.
37. W. Van Heddeghem, S. Lambert, B. Lannoo, D. Colle, M. Pickavet, and P. Demeester. Trends in worldwide ict electricity consumption from 2007 to 2012. *Computer Communications*, 50:64–76, 2014.
38. R. Verdecchia, F. Ricchiuti, A. Hankel, P. Lago, and G. Procaccianti. Green ICT research and challenges. In *Advances and New Trends in Environmental Informatics*, pages 37–48. Springer, 2017.

39. R. Verdecchia, R. Saez, G. Procaccianti, and P. Lago. *Empirical Evaluation of the Energy Impact of Refactoring Code Smells*. 5th International Conference on ICT for Sustainability. 2018.
40. S. G. S. A. U. Wilke, Claas; Richly. Energy proling as a service. In *Proceedings of INFORMATIK 2013, GI, LNI*, pages 1043–1052, 2013.
41. S. Zein, N. Salleh, and J. Grundy. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software*, 117:334–356, 2016.