SPELLing Out Energy Leaks
Aiding Developers Locate Energy Inefficient Code

Rui Pereira[b,1], Tiago Carção[b], Marco Couto[b], Jácome Cunha[c], João Paulo Fernandes[d], João Saraiva[b]

[a] *HASLab/INESC TEC & Universidade do Minho, Portugal*
[b] *C4 — Centro de Competências em Cloud Computing (C4-UBI), Universidade da Beira Interior, Rua Marquês d'Ávila e Bolama, 6201-001, Covilhã, Portugal*
[c] *NOVA LINCS & Universidade do Minho, Portugal*
[d] *CISUC & Universidade de Coimbra, Portugal*

Abstract

Although hardware is generally seen as the main culprit for a computer's energy usage, software too has a tremendous impact on the energy spent. Unfortunately, there is still not enough support for software developers so they can make their code more energy-aware.

This paper proposes a technique to detect energy inefficient fragments in the source code of a software system. Test cases are executed to obtain energy consumption measurements, and a statistical method, based on spectrum-based fault localization, is introduced to relate energy consumption to the source code. The result of our technique is an energy ranking of source code fragments pointing developers to possible energy leaks in their code. This technique was implemented in the SPELL toolkit.

Finally, in order to evaluate our technique, we conducted an empirical study where we asked participants to optimize the energy efficiency of a software system using our tool, while also having two other groups using no tool assistance and a profiler, respectively. We showed statistical evidence that developers using our technique were able to improve the energy efficiency by 43% on average, and even out performing a profiler for energy optimization.

1. Introduction

To detect inefficiency at runtime, many programming languages offer advanced profilers which locate source code fragments which are possibly responsible for such inefficiencies. In the same line of reasoning, while IDEs have traditionally incorporated powerful advanced type and modular systems, testing and debugging frameworks, and other tools to improve software developers

Email addresses: ruipereira@di.uminho.pt (Rui Pereira), tcarcao@di.uminho.pt (Tiago Carção), marco.l.couto@inesctec.pt (Marco Couto), jacome@di.uminho.pt (Jácome Cunha), jpf@dei.uc.pt (João Paulo Fernandes), jas@di.uminho.pt (João Saraiva)

productivity and effectiveness, there is no concrete evidence that this trend has included techniques to optimize or even analyze source code energy consumption [1, 2].

In fact, software developers are keen on developing energy-efficient software [1, 2], and a long list of (mostly recent) efforts that include [3, 4, 5, 6, 7, 8, 9, 10, 11, 12] have tried to provide developers with the libraries, tools, techniques and data to support energy-aware development. Even considering these efforts, the green computing research area is still at an early stage where research issues, challenges and opportunities abound [13, 14, 15]. Researchers [16, 17] also argue that there are two main problems in regards to energy efficient software development: **the lack of knowledge** and **the lack of tools**.

Indeed, if we compare energy-aware software engineering with the long lasting series of engineering techniques that aim at helping software developers quickly construct correct programs with optimal runtime we see an obvious deficit. While the latter includes compiler constructions such as partial and/or runtime compilation, advanced garbage collectors or parallel execution, the former is still clearly more modest in terms of achievements [14].

In the same line of reasoning, while IDEs have traditionally incorporated powerful advanced type and modular systems, testing and debugging frameworks, and other tools to improve software developers productivity and effectiveness, there is no concrete evidence that this trend has included techniques to optimize or even analyze source code energy consumption [1, 2].

This paper defines a technique, named SPELL - SPectrum-based Energy Leak Localization, that has been implemented in a tool, to determine *red* (energy inefficient) areas in software. The idea of this approach has been previously proposed in [18, 19, 20]. In this paper, we consider an *energy leak* synonymous to an energy inefficiency. In this context, a parallel is made between the detection of anomalies in the energy consumption of software during program execution, and the detection of faults in the execution of a program. Having this parallelism established, we adapted fault detection techniques, often used to investigate software bugs in program executions, to detect energy faults in programs.

The software system to be analyzed is executed with a set of test cases, and components of such system (for example, packages, functions, loops, etc.) are instrumented to estimate/measure the energy consumption at runtime. Inefficient energy consumption, the so-called energy leaks, are interpreted in SPELL as program faults, and we adapt Spectrum-based Fault Localization (SFL) techniques [21, 22] to relate energy consumption to the system's source code. Our analysis associates different percentage of responsibility for the energy consumed to the different components of the underlying system. Thus, the result of our analysis is a ranking of components sorted by their likelihood of being responsible for energy leaks, essentially pinpointing and prioritizing the developer's attention on the most critical *red* spots in the analyzed system. Thus, giving more useful information to have better support in making decisions of what parts of the system need to be optimized, ultimately helping place a new stepping stone for energy-aware programming.

Our proposed technique is language independent, allowing the analysis of

programs written in any programming language. Currently, it has been developed and focused on desktop and server systems only. A slight adaptation would be required to extend it into the mobile phone domain. Additionally, it is also context independent, allowing it to be applied to detect *red* areas on various levels of code. This means we could use it to detect the inefficiencies at different granularity levels, be that packages, classes, methods, functions, lines of code, etc. Even more so, the technique allows the use of different hardware component's energy values (CPU, DRAM, HDD, GPU, etc.) to compute the energy spent by a program, and may return the analysis of one specific factor (energy, time, or frequency of usage), or a global analysis considering all three factors.

Supported by our tool, our technique was able to identify potential energy leaks in the source code of concrete Java projects. Based on this identification, a set of expert Java programmers were then asked to improve the (energy) efficiency of those projects. The analysis of their success in doing so provided statistical evidence that the programs they ended up altering indeed consume less energy that the ones they were originally given, with an improvement, for different projects, between 15% and 74%.

Complementary, we compared the energy efficiency of the programs obtained as explained above against programs obtained from the original ones but by programmers working without the knowledge of any energy leak. From such comparison, we found statistical evidence that the difference is significant, in favor of the former: their performance is between 14% and 38% better.

A recurrent debate when optimizing energy consumption in software is whether a performance optimization is always an energy consumption optimization. Indeed, the `Energy` equation `(Energy = Power x Time)` does indicate that reducing time would imply a reduction in energy. However, the `Power` variable of the equation, not to be assumed as a constant, also has an impact alongside `Time`. Therefore, conclusions regarding this issue tend to diverge, where some works do support that optimizing for energy is optimizing for performance [5], while many others have studied contexts where the opposite was observed [4, 23, 24, 12, 25, 26, 27, 28, 29, 30]. This suggests that only looking at performance might not be enough for optimizing energy, and consequently performance profilers might also not be enough. Indeed we will show this is the case in Section 4.

In order to shed light and contribute to this debate with a particular focus on our context, we have complementary analyzed and compared our tool with an off the shelf profiler. This means that the experts were asked to improve the efficiency of the projects we considered with the guidance of SPELL and with the guidance of such profiler. Our analysis provided statistical evidence that experts with access to located energy leaks were able to better optimize the energy consumption of those projects than when using a profiler, with improvements between 2% and 72%.

The contributions of this paper are essentially four-fold:

- A language independent technique to locate energy inefficient components

in the source code of software systems. This technique is also independent of the approach used to measure (via external devices [3, 31, 32]) or estimate (via predictive models [33, 34]) energy consumption (Section 2).

- An implementation of our technique as a Java-based analysis tool (Section 3).
- An implementation of two (optional) auxiliary tools to facilitate: the energy measurements on Java programs (based on Intel's Runtime Average Power Limit (RAPL) technology [35, 31]), and the SPELL matrix construction. These are provided within the SPELL Toolkit along with the Java implementation of SPELL itself (Section 3).
- An evaluation of our technique and tool by detecting energy leaks in an empirical study. Programmers following SPELL recommendations were able to optimize programs to have energy gains of 43% on average (Section 4).

Furthermore, we discuss an overview of related research work in Section 5, and conclude our paper with final comments in Section 6.

2. Spectrum-based Energy Leak Localization

In this section we present our language independent technique, termed SPELL – or Spectrum-based Energy Leak Localization – that localizes *red* areas in source code. This technique combines energy measurements, program tracing, and a state-of-the-art fault localization technique, to detect source code components (such as methods) which are more likely to be responsible for abnormal or excessive energy consumptions. It follows a dynamic-oriented approach, i.e., it collects information of the software under analysis during its execution under normal usage scenarios or test cases.

We divided the definition of the technique into 4 parts. First, in Section 2.1, we thoroughly explain the fault localization concepts and the Spectrum-based Fault Localization technique (SFL). Building on such concepts, we then detail in Section 2.2 the changes performed to the core components of SFL, in order to collect information for each test/usage scenario regarding program tracings and energy consumption. Next, we define in Section 2.3 our concepts for how to analyze such information and reason about the energy impact of each source code component. Finally, to facilitate the explanation of our technique, we present a concrete example in Section 2.4, with all steps detailed.

Additionally, our technique is implemented within a Java toolkit, called the SPELL toolkit, which is presented in Chapter 3.

2.1. Spectrum-based Fault Localization

Our technique is based on spectrum-based fault localization [21, 22], a statistical analysis technique to detect faults in a program based on its implementation (source code).

In particular, SFL uses a hit spectrum (set of flags which reflect if a certain component is used or not in a particular run of the software) [21, 36] to build a matrix A of dimension $n \times m$, where m columns represent the different components (e.g. methods, classes) of a program during n independent test executions. A component can be anything being analyzed, be this a program, a package, a class, a method, or even a line of code. An entry $a_{i,j}$ in A of value 0 means that component j was not executed in test execution i, and an entry of value 1 means that it was. Complementing the hit spectrum, SFL also uses a vector e, with n elements, each of which indicates whether each of the n tests succeeded or not.

Equation 1 illustrates the generic format of A and e, and Equation 2 presents a concrete (simulated) example of the application of SFL with 3 test cases executed on a program with 4 components. The first line of the matrix A in the example, e.g., reads as: in the execution of the first test case, components c_1, c_2 and c_4 were executed and component c_3 was not. The first element of e, i.e., the value 0, indicates that the execution of the first test case met its expected output (or in other words, that it did not fail).

$$\begin{array}{c} \\ n \text{ spectra} \end{array} \begin{array}{c} \overset{m \text{ components}}{\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{bmatrix}} \end{array} \quad \overset{\text{error detection}}{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}} \tag{1}$$

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \tag{2}$$

Using (A, e), SFL tries to find which components are the most likely to be faulty by calculating: $n_{11}(j)$: the number of failed runs (indicated by the second 1 subscript) where component j was involved (indicated by the first 1 subscript); $n_{10}(j)$: the number of successful runs in which component j was involved; and $n_{01}(j)$: the number of failed runs where component j was not involved. This produces a $3 \times m$ matrix N, where m is the number of components in the program, and whose first/second/third line holds, for each component $j \in \{1, ..., m\}$, $n_{11}(j)$, $n_{10}(j)$ and $n_{01}(j)$, respectively.

Equation 3 shows the generic formulation of N and Equation 4 shows its instance for the illustration in Equation 2. Finally, SFL applies the Ochiai coefficient of similarity (Equation 5) to each component $j \in [1..m]$ to indicate which component has the highest probability of being faulty. This produces the matrix S given in Equation 6.

$$\overset{m \text{ components}}{\begin{bmatrix} n_{11}(1) & n_{11}(2) & \cdots & n_{11}(m) \\ n_{10}(1) & n_{10}(2) & \cdots & n_{10}(m) \\ n_{01}(1) & n_{01}(2) & \cdots & n_{01}(m) \end{bmatrix}} \tag{3}$$

$$\begin{bmatrix} 2 & 0 & 2 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} \tag{4}$$

$$S_j = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \cdot (n_{11}(j) + n_{10}(j))}} \tag{5}$$

$$\begin{array}{c} \text{4 components} \\ S \quad \begin{bmatrix} 0.82 & 0.0 & 1.0 & 0.5 \end{bmatrix} \end{array} \tag{6}$$

Analyzing the elements of matrix S, we finally conclude that component 3 is the most likely to be faulty. The rationale for this is that such component was involved in all the test executions that failed and was not involved in the test execution that succeeded.

In our proposed technique, that we introduce in the following subsections, we also rely on the spectrum of a program, which allows us to discriminate the usage of each component, and in what cases it was used, further extracting more information of the components being analyzed.

2.2. Static Model Formalization

Similarly to SFL, the technique that we propose, SPELL, relies on an input matrix A, with dimension $n \times m$, where the n lines also correspond to the number of test executions, and the m columns to the number of components.[1] It is very important to note that by *test* we mean test scenarios which replicate a real-world usage of the application, i.e., system tests. The quality of the tested scenarios is also important because only with tests which stress the components with different inputs replicating real-world scenarios, can one extract reliable information.

Differently to SFL, however, elements of A actually hold triples. Each such element $\lambda_{i,j}$ is defined as follows:

$$\begin{cases} (0, 0, 0), \text{ if component } j \text{ was not executed in test } i; \\ (E_{i,j}, N_{i,j}, T_{i,j}), \text{ otherwise.} \end{cases}$$

The execution data of each component is therefore segmented in 3 categories: E for energy consumption, N for the number of executions and T for the execution time.

In the energy consumption category, E, values of the energy consumed by different hardware components may be present, for example: CPU (E_{CPU}), DRAM (E_{DRAM}), fans (E_{fans}), HDD (E_{disk}), GPU (E_{GPU}), etc. At least one hardware component must be present.

The energy consumption values are expressed in the energy unit Joules (J), and the execution time is represented in milliseconds (ms). Finally, N holds the number of executions (cardinality).

[1] For a complete example please refer to Section 2.4.

2.3. Energy Leak Localization

Now that we have our spectrum model, we can begin extracting useful information and localizing the energy leaks.

While in SFL there is an error vector to reason about the validity of the output obtained by a test, the SPELL analysis does not receive an error vector. This is because there is still no known understanding to signal what can be seen as an excess of energy consumption. Therefore, an error vector needs to be calculated, and we define two different perspectives to calculate error vectors and similarities. These perspectives, that we describe next, are called *Component Category Similarity* and *Global Similarity*. An interesting consideration to draw here is that use of the error vector cannot result in a binary decision (pass or fail) for a test execution; the criterion has to use continuous values to represent the *greenness* of a test.

Component Category Similarity. The construction of this oracle was based on the regulation of greenhouse gas emissions for countries. After assessing how much is the total emission of gases in the different years, and depending on what each country contributed to these total emissions, each country is assigned a percentage of responsibility. We try to establish an analogy, where the n years are the different tests, and the m countries are the different components with the total for each category (energy, cardinality, and execution time), with the goal of assigning responsibilities to each component comparing with the total value.

To construct the error vector, we sum up all the values of all m components for each test $i \in \{1, ..., n\}$, shown in Equation 7:

$$e_i = (\sum_{j=1}^{m} E_{i,j}, \sum_{j=1}^{m} N_{i,j}, \sum_{j=1}^{m} T_{i,j}) \tag{7}$$

As this is applied for all tests, we obtain Equation 8, a vector of triples called e:[2]

$$e = [e_1 \ e_2 \ \dots \ e_n]^T \tag{8}$$

With (A, e) at hand, we now have an oracle model, and can begin localizing the energy leaks. Continuing our analogy of gas emissions, we need to relate the (3-category) data of each component with the total data. This is achieved *comparing* each component in A with e. The main goal is to obtain a simple structure containing the similarity between each column $j \in \{1, ..., m\}$ in A (which refers to the resources spent by component j) and vector e (the total amount of resources that were spent). This similarity can be interpreted as how much component j is responsible for each execution information of the total vector.

Assuming that $A(j)$ projects column j from matrix A, the similarity between component j and e is defined as ϕ_j, where:

[2]We use superscript T as the transpose of a matrix.

$$\phi_j = (\alpha_1(A(j), e), \alpha_2(A(j), e), \alpha_3(A(j), e)) \tag{9}$$

Finally, assuming that for $x \in \{1, 2, 3\}$[3], $A(j, x)$ and $e(x)$ project the x-th element from all the triples of $A(j)$ and e, respectively, we define:

$$\alpha_x(A(j), e) = \frac{\sum_{i=1}^{n} A(j, x)_i}{\sum_{i=1}^{n} e(x)_i} \tag{10}$$

To calculate the Ochiai coefficient similarity, we need to now be able to distinguish between a passed and a failed test. As previously stated, we cannot binarily define excess energy consumption. Thus, for this formula, we focused on the Jaccard similarity coefficient [37]. This coefficient is well-known and widely used to calculate the similarity coefficient between two vectors and has been used for a long period of time in the biology domain [38, 39], and is one of the most simple coefficients to implement. Using this definition, we calculate the similarity coefficient for each of the component's constituents E, N and T.

Applying this similarity function to all components $j \in \{1, ..., m\}$ will result in a row vector which represents, for each component and each test execution, their influence in the overall context for a given perspective (E, N or T). The higher the similarity (the closer it is to 1) the more responsible it is in that category.

Global Similarity. Using the similarity of each component category, we can have a parametrized analysis. However, it is also useful to have a value encoding the global similarity, allowing a numerical and global comparison between the different components.

The energy category E of a software component j can contain information on different hardware components such as *CPU*, *DRAM*, *GPU*, *fans*, and *disk*. These hardware components have different power consumption patterns that are known in advance. So, this information should be standardized according to the spontaneity/variation of those hardware components.

Let us assume the following scenario:

- For a concrete test suite, software components 1 and 2 showed the same total energy consumption;
- However, they rely differently on hardware components A and B, wherein A on average consumes more power than B;
- The energy of component 1 is only due to the use of component A;
- The energy of component 2 is only due to the use of component B;

[3]Here, indexes 1, 2 and 3 represent E, N and T respectively.

In spite of having the same consumption value, software components 1 and 2 should have their global similarity value influenced in different ways. As hardware component A has a higher average power consumption, component 1 is likely to contribute more to energy consumption than component 2 in scenarios that are not captured by the test suite in use.

A multiplicative factor can be defined for each hardware component and applied to allow standardization. Table 1 details the average power consumption for each component[4].

Table 1: Average W consumption for hardware components

Component name	**Average power consumption (W)**
CPU	102.5
DRAM	3.75
Fans	3.3
Hard Drive	7.5
GPU	187.5

Observing, e.g., that CPU is responsible for 34% of the total power consumption on average, for each test $i \in \{1, ..., n\}$ and component $j \in \{1, ..., m\}$ we propose the formula:

$$EF_{i,j} = 0.34 \times E_{\text{CPU}_{i,j}} + 0.01 \times E_{\text{DRAM}_{i,j}} + 0.01 \times E_{\text{fans}_{i,j}} + 0.02 \times E_{\text{disk}_{i,j}} + 0.62 \times E_{\text{GPU}_{i,j}} \tag{11}$$

Note this formula can be rewritten to account for any other combination of hardware parts (e.g., include a screen of a smartphone).

Now we can calculate the global value for each component:

$$\text{global}_c(j) = [\text{g}_c(1, j)\ \text{g}_c(2, j)\ \ldots\ \text{g}_c(n, j)]^T \tag{12}$$

where

$$\text{g}_c(i, j) = EF_{i,j} \times N_{i,j} \times T_{i,j} \tag{13}$$

This global value takes into consideration not only the energy consumption of a component, but the cardinality and execution time all as one value. This allows us to have a better understanding of what are the most important components to look at and try to optimize. For example, a component A may consume twice the amount of energy of component B, but component B is used five times as often which might make it a good candidate to prioritize the attention on. This would give a weight to component B as it would seem to be a core part of the analyzed program.

Once we have the global values for each component, we can proceed to calculate the global error vector as:

[4] `http://www.buildcomputers.net/power-consumption-of-pc-components.html`

$$\text{global}_e = [\text{g}_e(1)\ \text{g}_e(2)\ \ldots\ \text{g}_e(n)]^T \tag{14}$$

where

$$\text{g}_e(i) = \sum_{j=1}^{m} \text{g}_c(i,j) \tag{15}$$

Finally, we apply the similarity function α to each component j to obtain the global similarity with the error vector, defined as ψ,

$$\psi(j) = \alpha(\text{global}_c(j), \text{global}_e) \tag{16}$$

where

$$\alpha(c,e) = \frac{\sum_{i=1}^{n} c_i}{\sum_{i=1}^{n} e_i} \tag{17}$$

Once again, the higher the similarity value, and closer it is to 1, the more responsible it is. We can rank the components by this global similarity and have initial indicators of where in the program we should prioritize our attention on, and which are the most important components to optimize.

2.4. An Example

To understand how the SPELL analysis works and see how it handles the execution data, we present in the following a concrete example: a simple parking management system, containing 4 functionalities (add a car, add a list of cars, search for the oldest car, and sort cars by registration date). A code snippet expressing this example is depicted in Listing 1.

Listing 1: Pseudo-code for the Parking example

```
public class Park {

  List<Car> cars;

  void addCar(Car c) { cars.add(c); }

  public void addCars(Collection<Car> c) { cars.addAll(c); }

  public Car oldestCar() {
    Car r = null;
    if (!cars.isEmpty()) {
      r = cars.get(0);
    }
    for (Car c : cars) {
      if (c.getFirstRegistration() < r.getFirstRegistration()) {
        r = c;
      }
    }
    return r;
  }

  public void sort() {
    CarComparator c = new CarComparator();
    cars.sort(c);
  }

}
```

Considering that we want to perform the analysis at the method level, our software components will then be the methods. We consider four different methods/components: adding several cars at the same time, adding a single car, finding the oldest car, sorting the list of cars by registration date, and finding the oldest car after the list of cars is sorted. We can also consider five different test suites, each roughly simulating different (yet representable) usage scenarios for this program based on our methods/components.

We instrumented this program in order to collect, for each test, the energy consumed by each method (using RAPL). It also allowed us to obtain the usage frequency of each method, and its execution time. Therefore, after running the test suite, we can use the information of this program's execution and start the analysis.

Table 2: SPELL matrix built for the example program

	addCars	*sort*	*addCar*	*oldestCar*	e	$global_e$
t_1	$\begin{pmatrix}\{2.57, 0.74\}\\16\\264.0\end{pmatrix}$	$\begin{pmatrix}4.43, 2.26\\9\\692.0\end{pmatrix}$	$\begin{pmatrix}0.49, 0.07\\808\\47.0\end{pmatrix}$	$\begin{pmatrix}4.13, 2.34\\8\\638.0\end{pmatrix}$	$\begin{pmatrix}\{11.64, 5.43\}\\1641.0\\841.0\end{pmatrix}$	26972.3683
t_2	$\begin{pmatrix}\{1.48, 0.48\}\\8\\150.0\end{pmatrix}$	$\begin{pmatrix}3.86, 1.72\\14\\531.0\end{pmatrix}$	$\begin{pmatrix}0.92, 0.13\\1612\\76.0\end{pmatrix}$	$\begin{pmatrix}3.00, 1.62\\10\\414.0\end{pmatrix}$	$\begin{pmatrix}\{9.27, 3.97\}\\1171.0\\1644.0\end{pmatrix}$	53533.0614
t_3	$\begin{pmatrix}\{1.46, 0.42\}\\10\\152.0\end{pmatrix}$	$\begin{pmatrix}6.02, 2.41\\14\\672.0\end{pmatrix}$	$\begin{pmatrix}1.13, 0.18\\1626\\87.0\end{pmatrix}$	$\begin{pmatrix}1.02, 0.40\\2\\110.0\end{pmatrix}$	$\begin{pmatrix}\{9.65, 3.43\}\\1021.0\\1652.0\end{pmatrix}$	75171.9579
t_4	$\begin{pmatrix}\{1.22, 0.36\}\\8\\133.0\end{pmatrix}$	$\begin{pmatrix}0.56, 0.20\\2\\61.0\end{pmatrix}$	$\begin{pmatrix}0.16, 0.02\\160\\13.0\end{pmatrix}$	$\begin{pmatrix}1.14, 0.50\\4\\150.0\end{pmatrix}$	$\begin{pmatrix}\{3.10, 1.11\}\\357.0\\174.0\end{pmatrix}$	826.8864
t_5	$\begin{pmatrix}\{2.49, 0.64\}\\18\\259.0\end{pmatrix}$	$\begin{pmatrix}8.16, 4.19\\17\\1346.0\end{pmatrix}$	$\begin{pmatrix}0.66, 0.10\\1418\\82.0\end{pmatrix}$	$\begin{pmatrix}3.33, 1.64\\6\\468.0\end{pmatrix}$	$\begin{pmatrix}\{14.66, 6.59\}\\2155.0\\1459.0\end{pmatrix}$	98098.2886
ϕ	$\begin{pmatrix}\{0.191, 0.129\}\\0.0104\\0.151\end{pmatrix}$	$\begin{pmatrix}\{0.477, 0.527\}\\0.0097\\0.5204\end{pmatrix}$	$\begin{pmatrix}\{0.070, 0.025\}\\0.9747\\0.0481\end{pmatrix}$	$\begin{pmatrix}\{0.262, 0.318\}\\0.0052\\0.2805\end{pmatrix}$		
ψ	0.0375	0.4061	0.4970	0.0594		

We can see the entire model of the SPELL analysis for our example defined in Table 2, but let us construct it step by step. The input data can be seen in the top left $5 * 4$ matrix shown in Table 2, where each component and each test has a triple of three categories. This triple contains the CPU and DRAM energy consumption value, the number of times that software component was used, and the execution time:

$$\begin{pmatrix}E_{CPU}, E_{DRAM}\\N\\T\end{pmatrix}$$

In this case, the only hardware components shown are the CPU and DRAM for the sake of simplicity of presenting our technique, but it still straightfor-

wardly applies if energy consumption information of more hardware components is available.

Similarity by Component's Category. Having these inputs defined in SPELL, we will first calculate the software component similarities. We begin by building the error (e vector). To do so, for each test, we sum all the values of each individual category of the component data. This is shown on the right hand side of our example matrix under the e column. Next, we calculate each of the component's category similarity. For example, for the (CPU) energy category of component c_1 we will have the following:

$$\alpha_1(A(c_1), e) = \frac{2.57+1.48+1.46+1.22+2.49}{11.64+9.27+9.65+3.1+14.66} = 0.191$$

This would be applied for the both DRAM energy category and the other two categories, and for each of the other components, producing the results seen in the similarity by component's category row ϕ in Table 2.

Global Similarity. For the global similarity, we begin by calculating the global values of each component, and afterwards our new total global value vector. We obtain values $global_c(1)$ and $global_e$:

$$\text{global}_c(1) = [3722.1888\ 609.6\ 760.912\ 445.1776\ 3976.686]^T$$

$$\text{global}_e = [26972.3683\ 53533.0614\ 75171.9579\ 826.8864\ 98098.2886]^T$$

Finally, we use the coefficient similarity $\psi(1)$, to obtain the global similarity value for component c_1 of 0.0375. Applying this to each component, we obtain the results under the global similarity ψ.

Analysis. Having all the needed information to analyze this program we begin extracting useful information. Reading the global similarity values (ψ), we can see which component has the highest probability of having an energy leak with the order of *addCar* (with similarity of 0.4970), *sort* (with 0.4061), *oldestCar* (with 0.0594), and finally *addCars* (with 0.0375). This indicates to the developer that he should first consider looking into method *addCar* to try to improve the energy consumption of this program.

An advantage of this technique, which highlights the complementary perspectives of the two types of similarities that we consider, is that it can tell, besides having a global view of the component, indicators of why the component is faulty. For example, *addCar* is given the highest global similarity value, and *sort* the second highest. If we now look into their category similarity values (ϕ), we can see that although the former has the lowest (CPU and DRAM) energy similarity (0.070 and 0.025), it has the highest cardinality similarity by far (0.9747); the latter, however, has the highest energy similarity (0.477 and 0.527), but the second lowest cardinality similarity (0.0097). Even though *sort* has a higher impact in terms of energy consumption when compared to *addCar*, *addCar* is almost a core component of this example program, with a much higher usage than *sort* which ends up contributing to high energy consumption over the course of the program's lifecycle.

If the developer is only interested in optimizing purely for energy efficiency, *sort* would be the best place to start looking at. On the other hand, if the developer cares equally about all three categories, *addCars* should be looked at first according to our global similarity analysis.

We argue that when choosing the software component levels, the user must decide if they prefer a much more precise analysis, trading off performance, or the opposite.

Choosing the software component level to be analyzed will come at a trade-off of precision vs. performance. If the user has an idea of where a problem might be, they can focus their attention on a certain portion of the code. On the other hand, we argue that if the user has no clear indication of where to start, they should begin by package or class level components (based on program size), and continue to "drill-down" into finer granularity levels very much like how a profiler is to be traditionally used.

3. SPELL Toolkit

As previously stated, our technique is language independent, where the only required input is a matrix representing the tests, components, and categories. As a proof of concept we have implemented the SPELL technique in Java. To use SPELL in detecting energy leaks in software applications, we also provide two auxiliary (language dependent) tools which help automate the energy measurements within a Java program (using Intel's RAPL), and construct the SPELL matrix based on the measured outputs.

As the SPELL technique itself is language independent, one may easily develop front-end tools for other languages to measure the energy consumption and/or generate the SPELL matrix to run the analysis. Our core tool, and its two supporting tools are open source, and provided together within the SPELL toolkit. The toolkit[5] contains more information on how to run the tools, and the representation of the input and output data of each. An overview of the tools (solid blocks) within the SPELL toolkit is shown in Figure 1.

Instrumentation and Energy Measurements. The first auxiliary tool in the tool-kit, *Instrumentation*, consists of an out-of-the-box energy monitoring instrumentation tool which automatically instruments the source code of each method in a class with calls to the API of a Java energy estimation framework during the beginning and end of each method (including before any nested returns). When the instrumented program is executed, an output with both the execution trace and energy consumption is presented a file at the end of the execution. An example of the generated output trace is shown in the *Instrumentation* documentation.

This tool uses Intel's Runtime Average Power Limit (RAPL) [35], and the Java based RAPL framework jRAPL [40]. This allows us to record precise

[5]GitHub: `https://github.com/greensoftwarelab/SPELL`

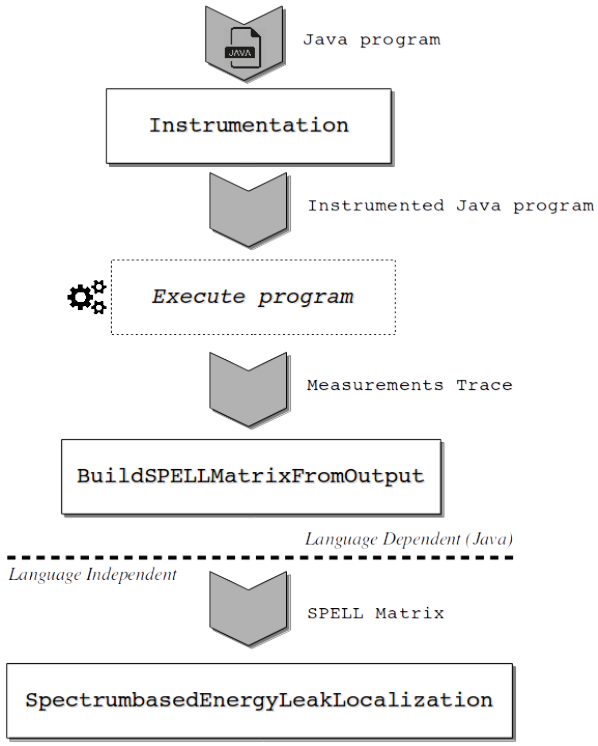

Figure 1: SPELL tool-kit contents overview

energy consumption measurements from several hardware components (CPU, DRAM, GPU, PACKAGE), as RAPL is a very reliable tool (as shown in [31] and [32]). In addition, the instrumentation itself is based on the JavaParser[6] set of tools to parse and instrument the code.

Note that SPELL is not limited to only using RAPL to measure energy, but is developed in a way that it may receive any energy measurement framework or tool to be used allowing the analysis of other languages. It also permits looking at other domains such as: Android applications when using Trepn [41];Monsoon [42], embedded devices using the ODroid XU-3 [7]; and other scientific works [27, 43, 44, 33, 34, 45, 46, 41, 47].

SPELL Matrix Construction. The second auxiliary tool, *BuildSPELLMatrixFromOutput*, uses the execution/measurement output log of our instrumented program as the input to construct a SPELL matrix. This automatically looks at the method calls, and aggregates the energy consumption, execution time, and

[6] JavaParser: `http://javaparser.org/`
[7] ODroid XU-3: `https://www.hardkernel.com/ko/tag/odroid-xu3/`

frequency of methods into our matrix representation of program components and tests.

If the previous tool (Instrumentation) were to be swapped out for another technology or measurement system, this tool would still create the correct SPELL matrix as long as its' input follows the same defined language. Nonetheless, this too can be switched for another SPELL matrix construction tool. An example of the generated output trace is shown in the *BuildSPELLMatrixFromOutput* documentation.

SPELL Analysis. The core tool within the tool-kit, *SpectrumbasedEnergyLeakLocalization*, is a Java implementation of the SPELL technique formally (and fully) defined in Section 2. This tool parses a SPELL matrix (for example the output of the *BuildSPELLMatrixFromOutput* tool), and calculates the Oracle, Similarity, and Global Similarity for all of the program's components given to be analyzed.

As SPELL is language independent, it does matter what programming language was analyzed nor with what measurement technique or tool. Any of the two auxiliary tools can be swapped by other similar tools if the user prefers a different approach, system, or domain.

4. Empirical Evaluation

One of our goals is to help provide programmers ways to become more energy-aware. Additionally, our SPELL technique is to be used by developers to help them detect energy leaks (or energy inefficiencies) on a source code level. Thus, we designed an empirical study to understand and answer the following research questions:

RQ 1 *Can the energy leaks identified by SPELL help developers improve the overall energy efficiency of their programs?*

Answering this question allows us to understand if in fact SPELL can detect areas in the source code where there is a probability of an energy hotspot occurring. If SPELL were to consistently point to areas in the code, where in turn the developer would go ahead and alter, and the energy efficiency improves, we can assume it is indeed identifying energy leaks. If it were to indicate areas where by the developer's changes actually brought about a deterioration in the energy consumption, then SPELL is not able to identify energy leaks.

RQ 2 *Are the programs improved by developers assisted by SPELL significantly more energy efficient than the programs improved by developers without tool-assistance?*

If a developer using SPELL is not significantly producing more energy efficient programs, then it would mean there is no need to use such a tool as a developer's own knowledge is enough for such a task.

RQ 3 *Are the programs improved by developers assisted by SPELL significantly more energy efficient than the programs improved by developers with an off the shelf profiler?*

This question is very important, as one might assume that SPELL is nothing more than another profiler, or that using an off the shelf profiler is enough to improve the energy efficiency of a program. Additionally, answering this question will allow us to understand if looking at a program's execution performance, and optimizing based off that information is enough to optimize for energy.

The following sections will describe in detail the design, execution, results and discussion of our empirical study.

4.1. Experimental Setup

Subjects. Participants in this study were selected from a candidate group that replied to an invitation that we publicized among our departments and two software houses. The selection process consisted of a self assessment step, in which to be eligible, candidates had to consider themselves experienced Java programmers. Ultimately, 15 programmers were selected: 12 male and 3 female; all with computer science background and/or professional experience: 6 postdoc researchers, 6 PhD students, and 3 professional programmers.

Design. For this study, we asked programmers to try to optimize the energy consumption of a program in three different scenarios: a control group, with our SPELL technique, and with a profiler.

The participants were then arranged into groups of threes (one for each scenario) according to their professional status. Essentially, the outcome was 5 different groups of 3: 2 groups of postdoc, 2 groups of PhD students, and 1 group of professional programmers.

Objects. In order to support the study, we initially considered 63 Java projects from an object-oriented course for computer science students, where students were asked to build a journalism support platform, where users (Collaborators, Journalists, Readers, and Editors) can write chronicles and reports, give likes and comments, and perform other tasks.

We filtered these projects to obtain the ones which passed a set of system tests designed by the course instructors, and all 16 unique operations and functional requirements were implemented (posting chronicles/reports, registering users, writing comments, viewing top commented, etc.). By doing so, we ended up with 42 comparable and differently implemented projects[8].

Due to allowing certain operations such as *Listing Comments*, and to provide an initial "warm-up", for each of the 42 projects we populated the system

[8] `http://www.di.uminho.pt/~jas/Research/spellStudies.rar`

with an initial set-up with: 3000 Chronicles, 3000 Reports, 7655 Likes, 8586 Comments, 60 Collaborators, 60 Journalists, 406 Readers, and 15 Editors.

To execute the projects, we defined 7 test scenarios (i.e., 7 scenarios replicating real program usage), simulating 7 days of interaction with the platform. Each test scenario was made up of a random number (varying between the hundreds and the thousands) of the 16 unique operations. While each test scenario contained each of the 16 unique operations, the randomness allowed certain *days* to have more of a certain type of operation than others. For example Tests 5 and 6 contain more write operations, while the others contain more read and lookup operations.

For selecting which projects would actually be explored in our study, we have resorted to SPELL itself. Indeed, we have used the test scenarios described previously to calculate the global similarity value for each of the 42 software projects (each component was defined as one project, so 42 components were analyzed in total). Project 1 (P_1) obtained a global similarities of 0.4259, P_{47} of 0.4093, P_{49} of 0.1439, P_6 of 0.0042, P_{59} of 0.0042, P_{36} of 0.0029, P_{17} of 0.0015, etc.

The reason for using SPELL here is that a higher global similarity represents a more probable scenario where an energy leak may be occurring as it is more responsible for the overall consumption, and means developers should focus their attention on that specific component as it is the most energy problematic one.

This gives us a ranking of the most problematic projects according to SPELL. However, still we do not know where to look at to try to optimize. Thus, applying SPELL to each program but considering components as methods would allow us to obtain a ranking of methods that are the most/least responsible for energy consumption. So, we ran the SPELL analysis on the 5 worst ranking projects, so that 1 project is considered by each of our participant groups, to localize where energy leaks are present on a method level.

The global similarity for each of the projects' methods where $\psi > 0.07$ or to show at least 2 methods per project is shown in Table 3. The first column indicates the project, while the second column states the problematic Class.method according to SPELL, and the third column states the global similarity value. The higher it is, the more responsible it is for the global inefficiency, and where a problem is most probable to be found.

As a profiling tool, we turned to the NetBeans (8.2) Profiler[9], a Java profiler integrated into the NetBeans IDE. By using the profiling methods mode, and more specifically the *Hot spots* tool[10], we were able to see what methods the tool was uncovering as performance bottlenecks. Presented in Table 3 are the methods pointed by the profiler, under the *Method (Profiler)* column, and under the *%* column is the percentage of time (CPU) of the method as stated by the *Hot spots* tools. Just as with SPELL, the higher the value, the more problematic the method is.

[9] `https://profiler.netbeans.org/`

[10] `https://profiler.netbeans.org/docs/help/5.5/snap_cpu.html`

Table 3: SPELL and Profiler ranking of methods from Projects P_1, P_{47}, P_{49}, P_6, and P_{59}. The first column represents the project number, the second and third the top methods and ψ reported by SPELL (as hotspots), and the last two represent the top worst methods and % reported by the profiler.

Proj.	Method (SPELL)	ψ	Method (Profiler)	%
P_1	voteInReport	0.97	voteInReport	95.3
	getUserLoggedInType	0.02	listArticlesByTheme	2.7
P_{47}	listAllChronicles	0.57	addComment	51.1
	listAllReports	0.15	listAllChronicles	15.8
	chronicleExist	0.12	chronicleExist	7.8
P_{49}	Like	0.27	ListTheme	29.3
	ListComments	0.19	ListTopic	27.5
	AddComment	0.10	ListComments	6.5
	ListTopic	0.08	Like	5.5
P_6	printNoticiaTopicoTema	0.40	listChronicles	32.8
	printCronicaTopicoTema	0.20	listReports	24.9
	isLogged	0.15	topChronicles	13.2
P_{59}	getArticle	0.94	getArticle	81.9
	vote	0.4	getComments	12.4

To further characterize the projects that we used, we show in Table 4 concrete metrics about them. Each line represents the metrics for a single project, with the last 3 being the minimum, average, and maximum values. Columns 2–4 are the number of classes, methods, and lines of code (LOC), respectively. Column 5 represents the max cyclomatic complexity present in that project from a single method. Finally, column 6 represents the average cyclomatic complexity for that class, excluding methods with a complexity of 0 or 1.

Table 4: Software Metrics for Projects P_1, P_{47}, P_{49}, P_6, and P_{59}

	Classes	**Methods**	**LOC**	**Max Comp.**	**Avg Comp.**
P_1	38	181	1037	26	5.05
P_{47}	32	155	923	25	2.38
P_{49}	27	131	811	17	3.37
P_6	15	122	691	37	5.04
P_{59}	32	151	905	11	3.45
Min	15	122	691	11	2.38
Avg	28.8	148	873.4	23.2	3.86
Max	38	181	1037	37	5.05

Measurements. In order to analyze the energy consumption of all projects, we have instrumented their code using the SPELL toolkit. The instrumentation code is realized with calls to RAPL, which allows us to measure and monitor the energy that is being consumed.

Our measurements were made on a desktop with the following specifications: Linux 3.13.0-53-generic operating system, with 8GB of RAM, and a

Sandy Bridge Intel(R) Core(TM) i3-3240 CPU @ 3.40GHz. In the architecture of our machine, RAPL is only able to provide information regarding the energy consumption of the CPU. Each test was executed 30 times [48], and we extracted the cardinality and average values for both the time and CPU energy consumption (of the specific test and not the initial population as to only analyze the tests).

4.2. Execution

We asked our 5 groups of participants to analyze one of the 5 projects and, to the best of their knowledge, optimize its energy performance. Each group was randomly assigned one of the 5 projects. They were also given the project's description and input examples to familiarize themselves with the software requirements and structure, and allowed them to navigate the program looking at whatever they felt they needed to understand. We asked them to dedicate approximately 30 minutes to first understand the project. Each participant was given a series of test cases and their expected outputs. This allowed them to verify if they changed the business logic when refactoring and optimizing the project.

Finally, we randomly chose one of the participants in each group to have access to information produced by our SPELL technique for the given project, and one to have access to information produced by the NetBeans profiler. Both were asked to closely follow the recommendations of the tools. Thus, for each group/project, one participant used SPELL, one used a profiler, and one used no tool (control-group). The only imposed restriction was to try to dedicate no more than 2 hours to optimize the project.

We instructed them to take note of the time they began and, when they were satisfied with their work and felt they did indeed made an impact to the performance, to take note of the end time. They were also asked to describe what changes they made (or, if due to time restrictions, what changes they would make), and if they (non control-group participants) found it beneficial to have the data produced by the tools when optimizing for energy, or if they (control-group participants) would have found it impactful.

Afterwards, we collected all the refactored programs (3 different variations for each), made sure everything produced the expected output, and measured the energy consumption and execution time from these refactorings.

4.3. Results

Table 5 present the results for Projects P_1, P_{47}, P_{49} P_6, and P_{59}, respectively. Each row under *Test* represents the data for one of the 7 tests scenarios, with the final row being the totals and global values. The first block of 2 columns represents the data for the original project, showing Joules (J) and execution time in milliseconds (ms). The second, third, and fourth block (with 4 columns each) represent the measured energy, execution time, and energy gain percentage (relative to the original project) for the control group, SPELL group, and profiler group, respectively. The time taken to optimize is shown in parentheses above

each block next to the group name. A graphical representation of the global percentage of gains for each project can be seen in Figure 2, where the blue dotted bars represents the energy improvement (Joules) and the orange bars represent the execution time improvement (ms).

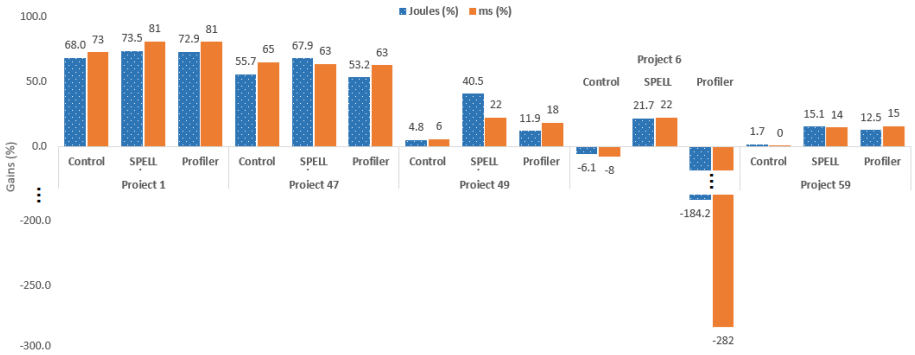

Figure 2: Global percentage of gains for all projects

4.4. Discussion

To validate improvements and changes in energy consumption, we tested the following hypothesis:

$$H_0 : P(A > B) = 0.5$$

$$H_1 : P(A > B) \neq 0.5$$

where $P(A > B)$ represents, when we randomly draw from both A and B, that the probability of a draw from A is larger than the one from B is 50% in the case of our null hypothesis, and different than 50% in our alternative hypothesis. To understand if there is an overall significant relevance between the (A,B) distributions, and not only per test scenario or per project, the data from all 30 measured samples, 7 tests, and 5 projects were grouped per distribution (Original, Control, SPELL, and Profiler). The distributions were defined in the following (A, B) pairs: (Original, Control), (Original, SPELL), (Original, Profiler), (Control, SPELL), and (Profiler, SPELL). We consider the samples as independent, non-normal distributed, and ran the Wilcoxon signed-rank test with a two-tail P value with α=0.01. The improvements were indeed very significant, producing significant relevance in all 5 cases, with p-values < 0.0001.

To calculate a nonparametric effect size, Field [49] suggests using Rosenthal's formula [50, 51] to compute a correlation, and compare the correlation values against Cohen's [52] suggested thresholds of 0.1, 0.3, and 0.5 for small, medium, and large magnitudes respectively. Thus we obtained the values of: 0.4 (medium), 0.6 (large), 0.3 (medium), 0.6 (large), and 0.5 (large) for the respective 5 (A,B) pairs. Thus, we can see that SPELL outperforms the profiler when compared to both the original versions, where SPELL achieved a large

Table 5: Study results from all projects

		Original		Control - (2h05)				SPELL - (1h13)				Profiler - (1h33)			
						Gain (%)				Gain (%)				Gain (%)	
	Test	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms
P_1	1	93.1	8621	17.8	1289	80.9	85	13.9	913	85.0	89	13.8	888	85.2	90
	2	20.3	1645	11.3	1796	44.2	-9	8.8	537	56.4	67	9.3	567	54.0	66
	3	87.4	7982	15.7	1146	82.0	86	13.8	869	84.2	89	13.8	869	84.2	89
	4	32.0	2666	15.0	1005	53.0	62	13.4	859	58.0	68	14.0	905	56.2	66
	5	58.5	5322	14.9	985	74.5	81	11.8	784	79.8	85	11.9	785	79.6	85
	6	17.9	1343	15.0	753	16.1	44	11.7	725	34.6	46	12.1	753	32.2	44
	7	14.0	928	13.6	850	2.9	8	11.9	725	14.8	22	12.6	780	10.1	16
	Total	323.1	28507	103.3	7824	68.0	73	85.5	5413	73.5	81	87.6	5547	72.9	81

		Original		Control - (2h02)				SPELL - (1h16)				Profiler - (0h44)			
						Gain (%)				Gain (%)				Gain (%)	
	Test	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms
P_{47}	1	51.4	4487	19.3	1216	62.3	73	13.3	1160	74.1	74	21.5	1417	58.2	68
	2	18.2	1235	10.3	641	43.2	48	7.6	798	58.4	35	10.1	643	44.2	48
	3	36.7	2972	14.4	899	60.8	70	11.1	1018	69.8	65	16.1	1022	56.2	66
	4	44.3	3683	18.1	1197	59.2	68	11.2	1024	74.7	72	19.1	1268	56.8	66
	5	39.3	3323	18.3	1266	53.5	62	14.1	1267	64.1	61	18.8	1270	52.3	62
	6	26.9	2024	15.6	991	42.1	51	13.0	1166	51.7	42	16.0	1008	40.4	50
	7	30.0	2311	13.3	836	55.5	64	8.9	882	70.3	61	13.9	881	53.5	62
	Total	246.7	20034	109.3	7045	55.7	65	79.1	7316	67.9	63	115.5	7510	53.2	63

		Original		Control - (1h49)				SPELL - (0h47)				Profiler - (0h36)			
						Gain (%)				Gain (%)				Gain (%)	
	Test	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms
P_{49}	1	40.2	2508	39.8	2458	1.1	2	23.4	2085	41.9	17	33.5	2056	16.8	18
	2	19.9	1392	17.1	1210	14.2	13	9.0	972	54.8	30	15.9	943	20.0	32
	3	34.1	2094	33.4	2042	2.1	2	15.8	1539	53.7	27	29.8	1728	12.6	17
	4	36.0	2202	36.0	2200	0.1	0	17.0	1701	52.7	23	31.7	1865	12.0	15
	5	24.5	1572	22.0	1380	10.1	12	20.2	1280	17.5	19	22.9	1341	6.4	15
	6	19.9	1240	19.1	1182	4.0	5	17.1	1092	13.9	12	19.1	1063	3.9	14
	7	29.3	1813	26.8	1644	8.5	9	18.8	1289	36.0	29	26.8	1501	8.7	17
	Total	203.9	12821	194.1	12115	4.8	6	121.3	9958	40.5	22	179.7	10497	11.9	18

		Original		Control - (2h13)				SPELL - (1h22)				Profiler - (2h00)			
						Gain (%)				Gain (%)				Gain (%)	
	Test	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms
P_6	1	18.5	1351	19.0	1460	-2.6	-8	12.9	966	30.2	28	79.7	7781	-330.9	-476
	2	9.4	600	10.3	668	-9.8	-11	7.8	487	17.0	19	13.9	1072	-47.9	-79
	3	13.0	878	14.2	969	-9.8	-10	9.8	663	24.2	24	33.7	3010	-160.6	-243
	4	21.2	1519	21.7	1571	-2.1	-3	17.1	1215	19.5	20	72.4	6953	-240.9	-358
	5	13.1	939	14.8	1061	-13.0	-13	10.7	732	18.2	22	18.1	1453	-37.8	-55
	6	12.0	804	13.2	902	-10.3	-12	10.3	673	14.3	16	14.0	1010	-16.3	-26
	7	18.7	1254	19.1	1306	-2.2	-4	14.3	986	23.6	21	69.4	6759	-270.5	-439
	Total	106.0	7345	112.4	7937	-6.1	-8	83.0	5723	21.7	22	301.2	28038	-184.2	-282

		Original		Control - (1h58)				SPELL - (1h04)				Profiler - (1h21)			
						Gain (%)				Gain (%)				Gain (%)	
	Test	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms
P_{59}	1	13.2	989	13.3	992	-0.6	0	11.0	803	16.5	19	11.3	797	14.9	19
	2	8.0	453	7.1	436	11.5	4	5.5	391	31.1	14	6.6	395	18.0	13
	3	10.2	722	10.4	730	-1.8	-1	8.5	643	16.6	11	9.4	630	8.3	13
	4	10.5	763	10.4	758	1.1	1	9.8	679	6.8	11	9.5	648	10.0	15
	5	11.5	840	11.5	846	-0.5	-1	9.8	681	14.9	19	10.0	687	12.5	18
	6	10.0	633	9.3	611	7.0	3	8.6	548	14.1	13	8.4	539	16.5	15
	7	7.8	529	8.0	535	-2.5	-1	7.3	472	7.0	11	7.3	472	7.0	11
	Total	71.3	4929	70.1	4908	1.7	0	60.5	4215	15.1	14	62.4	4168	12.5	15

effect size and the profiler a medium effects size, and to each other with also a large effect size.

Returning to our research questions, we have shown that there is both significant relevance and a large effect size when using our SPELL technique to improve the energy efficiency of programs, with an average energy gain of 44% (**RQ1**). While both the control-group (no tool assistance) and the profiler group did also produce significant relevance with their energy optimizations when compared to the original versions, SPELL outperformed both. Whereas the control group achieved a medium effect size, SPELL achieved a large effect size and when comparing SPELL to the control group, the former once again achieved a

large effect size (**RQ2**). Finally, the same applies to the profiler group where it achieved a medium effect size when comparing the optimizations to the original version (versus the large effect size of SPELL), and again SPELL achieved a large effect size when comparing to the profiler group (**RQ3**).

Additionally, we conducted a study to understand what the control group developers did to their projects, why did they end up producing gains, and why were those gains always lower when compared to the SPELL group. The goal was to understand if there were potential energy issues that SPELL was not able to detect. We observed that for Project P_1 the developer in the control group focused on modifying 3 methods, 2 of which ranked last by SPELL, while the other was ranked first. For Projects P_{47} and P_{49}, developers in the control group changed several methods, which according to SPELL were not the most problematic. For example, in P_{47}, the users modified the 3^{rd}, 4^{th}, 8^{th}, 9^{th}, and 23^{rd} most problematic methods according to SPELL, and in P_{49}, they modified the 5^{th}, 7^{th}, 10^{th}, 17^{th}, 22^{nd}, and 25^{th} most problematic ones. Thus, while these control group users were able to indeed optimize the program as they tackled methods which had issues (and were also identified by SPELL), they tackled methods which were not the most critical as the higher SPELL ranked methods. Finally, the control group developer for Project P_6 chose to modify the main method, by replacing the mechanisms used to process input by a new class he created, and the one for P_{59} actually only replaced global imports (e.g. `java.util.*`) by specific ones (e.g. `java.util.HashMap`) These last two ones explained why they became worse.

Observations From this study, we can see several interesting observations. In the case of Project P_1 and P_{59}, the rankings from both using SPELL and the profiler pointed to the same principal method (as shown in Table 3). Both were given a very high responsibility percentage (by SPELL) and high CPU time (by the profiler). This meant that if this method was optimized, a great impact in the performance would occur as this was, without a doubt, a very problematic method due to a bottleneck. Consequently for these two projects, the participants achieved very similar energy optimizations as one would expect. The slight difference can be attributed to what methods SPELL and the profiler pointed to afterwards, with the SPELL recommendations producing slightly better results.

We can also see how programmers with access to the SPELL recomendations were more efficient spending between 38%–57% less time, compared to the control-group, to detect and correct the problem, while also producing more efficient programs in both cases of energy and execution time. While those with the profile recomendations did also spend less time, they did not achieve results as good as those with the SPELL recommendation as we have seen. The participants also felt that having the ranking of responsibility percentage was very useful in identifying the *energy leaks* in the code, while the participants without this information expressed how they did not know where to start looking, or if certain parts were in fact problematic. All this is actually what we expected (for both SPELL and the profiler) as there is a substantial impact on having tools for energy-aware programming, as also suggested by [1, 2].

Moreover, when comparing the obtained gains from the control group with the ones from the SPELL group, we can conclude that theres is not much left out by SPELL in terms of energy leaks or energy inefficient methods. Considering the changes made to the code by control group developers, we observed that when they modified methods with highest ranking assigned by SPELL, than the obtained gains of close to the ones in the SPELL group. This happened in Project P_1, where both the control and SPELL groups changed the method ranked #1 by SPELL, and respectively obtaining gains of 68% and 73.5%.

We also observed that the gains obtained by the control group are somewhat proportional to the SPELL ranking of the modified methods, as demonstrated by the results for projects P_{47} and P_{49}. Additionally, as demonstrated by the results of P_6 and P_{59}, when the changes do not reflect the SPELL output, than energy consumption either stays roughly the same (as in Project P_{59}), or even increases (as in Project P_6).

Another interesting case is in Project P_6, where the results indicate a clear efficiency loss (both time and energy) for the case study using the profiler information. By comparing the original and transformed versions of the code, we discovered that the programmer responsible for this study decided to optimize the code by improving the efficiency of all listings and lookups on data structures, hence worsening insertions. The fact is that the feature tests that we provided contained more insertions than listings or lookups, leading to a decrease in the refactored version's performance. To understand if this outlier skewed our previous statistical analysis, we re-ran the analysis without considering Project P_6. The results maintained the same, with the only difference being the profiler obtained a slightly larger effect size when compared to the original projects. Thus, this does not change the conclusions of the study.

As the study only focused on giving participants one "round" or iteration of both the SPELL and profiler analysis, the participants using SPELL and the profiler tended to be "satisfied" with their optimizations much quicker, with time to spare in their maximum 2 hours scenario. In a real-world scenario, they would then run through another analysis, looking for new (if any) *energy leaks* and continue to further optimize if possible.

Finally, none of the participants had any knowledge of what techniques or optimizations could be done to specifically reduce energy consumption before going into the study. Nevertheless, with the knowledge on basic performance issues, algorithms, program complexity, and generally aiming for standard execution time optimization, they were able to achieve good results.

4.5. Looking back with DRAM

In this paper, we have so far provided evidence that SPELL helps developers to identify the components of a software program that can be improved to gain energy efficiency.

So far, however, and although we have argued that SPELL can receive inputs from different hardware components, we have only shown its effectiveness when using CPU measurements. This was due to the fact that, when we have initiated

this research direction, only energy measurements from the CPU were available in the machines we targeted. Since then, we were able to use an upgraded server, which allows us to access both CPU and DRAM measurements.

In the remainder of this section, we re-executed the initial stages of our study to calculate the global similarity considering these two hardware components. Our purpose for doing so is twofold. For once, we seek to understand what impact the DRAM energy consumption would have had on our study. On the other hand, we also aim to validate the consistency of SPELL across different systems.

The steps, and methodology we followed here are identical as before. The measurements were made on a new system with the following specifications: Linux Ubuntu Server 16.10 operating system, kernel version 4.8.0-22-generic, with 16GB of RAM, a Haswell Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz.

Table 6: SPELL with and without DRAM ranking of methods from Projects P_1, P_{47}, P_{49}, P_6, and P_{59}

Proj.	Method (SPELL)	ψ	ψ	Method (SPELL w/ DRAM)
P_1	voteInReport	0.97	0.99	voteInReport
	getUserLoggedInType	0.02	0.00	getUserLoggedInType
P_{47}	listAllChronicles	0.57	0.52	listAllChronicles
	listAllReports	0.15	0.16	listAllReports
	chronicleExist	0.12	0.1130	Like
	Like	0.078	0.1125	chronicleExist
P_{49}	Like	0.27	0.35	Like
	ListComments	0.19	0.13	ListComments
	AddComment	0.10	0.09	AddComment
	ListTopic	0.08	0.07	ListTopic
P_6	printNoticiaTopicoTema	0.40	0.40	printNoticiaTopicoTema
	printCronicaTopicoTema	0.20	0.22	printCronicaTopicoTema
	isLogged	0.15	0.17	isLogged
P_{59}	getArticle	0.94	0.93	getArticle
	vote	0.04	0.05	vote

The global similarity results are shown in Table 6. The left-hand side are the results from the original analysis (also shown in Table 3), and the right-hand side are the results from the SPELL analysis including DRAM energy consumption. In almost all cases, the rankings between both analyses maintained the same, with slight differences in the global similarity (ψ) value. The single exception can be observed in Project P_{47}. Here, the `Like` method came in fourth with a ψ value of 0.078, while it came in third with a ψ value of 0.1130 just slightly surpassing the `chronicleExist` method when DRAM energy consumption was also analyzed.

The initial hypothesis was that the results from the first analysis would not suffer any major changes in this case, as DRAM does not tend to have a high impact in overall energy consumption, as shown in other research [30, 53].

Even so, this post-analysis shows how having more available information on the energy consumption of different hardware components (for example, DRAM) can bring about a deeper analysis, and such as in the case of Project P_{47}, can reveal more information on the problematic spots within one's application. The more components considered, the more accurate of an analysis can be performed by SPELL.

4.6. Threats to Validity

We present now some threats to validity of our study, divided in four categories as defined in [54].

Conclusion Validity From our experiment it is clear that we can effectively find energy *hot spots* in source code, both on a project level, and on a method level. Moreover, through the empirical study we have shown that these results are useful for programmers. Nevertheless, by energy consumption we only considered energy consumption that can be related to CPU usage due to our machine limitations. While we have shown that energy and performance are sometimes related in non-predictable ways, the impacts of other hardware components on energy consumption deserve further elaboration. Thus, we intend to explore this in the future by running a similar study on a machine with a more recent architecture.

Internal Validity In this case we are concerned with other factors that may interfere with our experiment results. The energy consumption measurements we have for the different projects could have other factors than not just the source code itself. To avoid this we ran all the tests in the same way. For every test we added a "warm-up", and we ran every test 30 times, taking the average values for these runs so we could minimize particular states of the machine used and its other software. Also, the results from participants may have been influenced by other factors other than the SPELL and profiler recommendations we gave them. However, the results achieved through the five projects are quite consistent.

Construct Validity The purpose of our study was to evaluate our SPELL technique alongside programmers, to both properly understand the benefits of our technique with programmers, and to validate the efficiency of our technique in detecting energy leaks. Thus, we constructed an empirical study based off the suggestions of Ko et al. [55]. For example, for the task duration, they suggest that the tasks should not be so easy as to have almost every participant complete them before the time expires (leading to *ceiling effects* [56]), nor making it so difficult that no one can complete them in the allotted time no matter which tool is used (leading to *floor effects* [56]). Both of these cases would make it difficult to statistically discriminate and show the differences between tools.

Due to this, and in addition to another suggestion that such studies should not be more than 2 hours long [55], we decided to use the academic Java projects we presented. This allowed us to have projects which were neither too difficult nor too easy to both understand and optimize within our established time limit. Using larger real-world applications would introduce a risk of participants not completing or understanding (possible *floor effects*) due to the complexity and

possible lack of domain documentation. Nevertheless, there is no basis to suspect that these projects are better or worse than any other kind we could have used.

External Validity In this case we are concerned about the generalization of the results. The used source code has no particular characteristics that could influence our findings. Its only particularity is that it is written in Java, and maybe different results could be found for other PLs. However, our technique is independent of the language and thus we do not anticipate that. Thus, we believe that these results can be further generalized for other programs.

5. Related Work

While green computing exists for at least a decade, only recently has it started to trend with the growing concern of the impact on our environment. In average, close to 50% of the energy costs of an organization can be attributed to the IT departments [57]. Researching and designing energy-aware programming languages is an active area [58]. In fact, programmers many times seek help in resolving energy inefficiencies, showing that there are many misconceptions within the programming community as to what causes high-energy consumption, how to solve them, and a heavy lack of support and knowledge for energy-aware development [1, 2, 59, 60], greatly motivating this work. This awareness of energy consumption is notorious within the software testing area, where some works aim at reducing the overall consumption in the testing phase, by reducing the number of tests while maintaining the code coverage [8, 61].

Studies have shown how different design patterns [62, 63], sorting algorithms [64, 65], android API and advertisements [6, 7, 66, 67], software version changes [10], code obfuscations [68], refactorings and transformations [9, 69, 70], and different Java based collections [29, 71, 72, 73] have a statistically significant impact on energy usage. Studies have also shown how different programming languages have very different energy usages [24, 30, 74]. Other researchers have used a model-based power consumption analysis in Android mobile applications [75, 76, 77, 78].

While we measured our programs using Intel's RAPL framework, there are other possible ways of measuring or estimating energy consumption [27, 43, 44, 33, 34, 45, 46, 41, 47]. As SPELL is not dependent on a specific measurement technique, these measurement techniques can easily replace RAPL and, alongside SPELL, reason about the energy measurements to present a target area of where one should focus their attention to optimize.

It is common for software developers to use debugging tools and profilers to help detect bugs or performance inefficient code fragments. Applying these concepts to help detect energy inefficient code fragments is a much more challenging task. There is still very little knowledge as to what can be directly done, from a software developers position, to manipulate and improve energy consumption. Even if a developer takes the steps and effort to use one of the many energy/power measuring devices, a lot has to be taken into account such as the contextual information about what the program is supposed to be doing,

or where it was executed. Thus, this challenging problem has attracted several researchers to propose solutions, but with a focus on mobile applications.

Ma et al. [76] presented a tool, *eDoctor*, for mobile users to troubleshoot any irregular battery draining issues they were having on their smartphones. The authors' tool analyzes a mobile application's behavior, and identify abnormalities. It then suggests the user the most appropriate repair solutions, such as disabling device locations, downgrade to previous versions, turn on airplane mode, etc. A different approach was done by Oliner et al. [79], where a black-box diagnostic is performed. The client application sends coarse-grained measurements to a server where the data is correlated with client properties (for example running applications). It then suggests actions the user may make on the mobile phone to improve battery life.

Linares-Vásquez et al. [6] conducted a large empirical study on API calls and usage patterns, within the Android development framework, to find which have a tendency to have high consumption costs. Their study was conducted on 55 different apps, looking into 807 different API methods and defined 131 as *energy-greedy APIs*. Similarly, Liu et al. [80] analyzed 402 different Android applications and found that there were two main causes of energy problems: missing deactivation of sensors or wake locks, and cost ineffective use of sensory data. In response, they developed *GreenDroid*, a tool to identify these two problems to further help developers find these issues.

Two similar and complementary works [67, 81], also within the Android domain, defined energy efficient guidelines for mobile development. The former was based on performance guidelines for mobile and focused on code smells affecting CPU usage. The latter focus on resource usage, leakage, and sensors.

These prior works, while having the same objectives as our *SPELL* techinique, are based on previously known energy guidelines. They focus on finding the patterns, bugs, API, etc., and point to these areas. We believe that the works which relates the most to our own are the following two.

Couto et al. [82] presented a technique where they relate the energy consumption to the source code of the application while giving classifications of methods as Red, Yellow, or Green. They do so by running each test case twice on the program, where first they log the stack trace of each test, and then they log the energy values for a test. By correlating the stack trace with the energy values, and using thresholds, they classify the tests as Red, Yellow, or Green. Finally, depending on what methods were called in those tests, also classify each test as Red, Yellow, or Green. With our technique, as we use measured values for each component, we can provide a more detailed and fine-grained analysis. Our technique is also not limited in its scope, by this we mean we can easily analyze code-line, method, class, package level etc., as our technique analyzes the components which has no restriction on its definition, and is also language independent.

Recently, Verdecchia et al. [83] presented a naive spectrum-based fault localization technique aimed to efficiently locate energy hotspots in source code. Their work is very closely related to ours. The authors state that the difference between their work and ours is while our contribution lies more in providing

the means to precisely locate energy hotspots in source code, their work aims to investigate if more naive approaches can be used to locate them. Thus, understanding both sides, research can be further done on finding the best balance of performance and precision.

6. Conclusion

This paper introduced SPELL — a spectrum-based energy leak localization technique to identify inefficient energy consumption in the source code of software systems. This technique uses a statistical method to associate different percentage of responsibility for the energy consumed to the different source code components of a software system, thus pinpointing the developer's attention on the most critical *red* spots in his code. Such software components may not only be source code fragments, but also a set of equivalent software systems from which we need to select the greenest one.

As future work, we plan on adapting, evolving, and testing our technique on a mobile phones, as currently it is only for desktop and server based systems.

The paper also presented the implementation of this technique as a language independent tool to locate energy leaks in a program's source code. A front-end for the Java language was constructed to monitor energy consumption at runtime, which uses the developed SPELL tool to locate leaks in Java.

To evaluate both our technique and tool, we executed an empirical study where we asked five groups of three developers to optimize the energy efficiency of a software system. One developer had no tool assistance, while the other two used our SPELL technique and a profiler, respectively. We showed that developers using our technique were able to improve the energy efficiency of their programs by 43% on average, while also showing statistical evidence that the difference between a profiler and our technique is significant, in favor of the former: the performance is between 2% and 72% better.

Thus, we have also shown that optimizing for energy efficiency is not directly the same as optimizing for performance. We also showed that using our technique, the performed optimizations achieved on average a lower *Powerup* (implying average power savings, with better performance and energy efficiency), while optimizations following a profiler's recommendations achieved better performance at the cost of energy efficiency.

Acknowledgements

This work is funded by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalization - COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI010145FEDER-016718, UID/EEA/50014/2013, and by FCT grant SFRH/BD/132485/2017. This work is also supported by operation Centro010145FEDER000019 — C4 —

Centro de Competências em Cloud Computing, cofinanced by the European Regional Development Fund (ERDF) through the Programa Operacional Regional do Centro (Centro 2020), in the scope of the Sistema de Apoio à Investigação Científica e Tecnológica - Programas Integrados de IC&DT, and the first author was financed by post-doc grant referência C4_SMDS_L1-1_D.

References

[1] G. Pinto, F. Castor, Y. D. Liu, Mining questions about software energy consumption, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 22–31.

[2] C. Pang, A. Hindle, B. Adams, A. E. Hassan, What do programmers know about software energy consumption?, IEEE Software 33 (3) (2016) 83–89.

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

[4] G. Pinto, F. Castor, Y. D. Liu, Understanding energy behaviors of thread management constructs, in: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, ACM, 2014, pp. 345–360.

[5] T. Yuki, S. Rajopadhye, Folklore confirmed: Compiling for speed= compiling for energy, in: Languages and Compilers for Parallel Computing, Springer, 2014, pp. 169–184.

[6] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, D. Poshyvanyk, Mining energy-greedy api usage patterns in android apps: an empirical study, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 2–11.

[7] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, P. Ammann, Ecodroid: An approach for energy-based ranking of android apps, in: Proceedings of 4th International Workshop on Green and Sustainable Software, GREENS '15, IEEE Press, 2015, pp. 8–14.

[8] R. Jabbarvand, A. Sadeghi, H. Bagheri, S. Malek, Energy-aware test-suite minimization for android apps, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, 2016, pp. 425–436.

[9] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, ACM, New York, NY, USA, 2014, pp. 36:1–36:10. doi:10.1145/2652524.2652538. URL http://doi.acm.org/10.1145/2652524.2652538

[10] A. Hindle, Green mining: a methodology of relating software change and configuration to power consumption, Empirical Software Engineering 20 (2) (2015) 374–409.

[11] S. Li, S. Mishra, Optimizing power consumption in multicore smartphones, Journal of Parallel and Distributed Computing 95 (2016) 124–137.

[12] L. G. Lima, G. Melfe, F. Soares-Neto, P. Lieuthier, J. P. Fernandes, F. Castor, Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language, in: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'2016), IEEE, 2016, pp. 517–528.

[13] A. E. Trefethen, J. Thiyagalingam, Energy-aware software: Challenges, opportunities and strategies, Journal of Computational Science 4 (6) (2013) 444 – 449.

[14] P. Lago, Challenges and opportunities for sustainable software, in: Proceedings of the Fifth International Workshop on Product LinE Approaches in Software Engineering, PLEASE '15, IEEE Press, 2015, pp. 1–2.

[15] A. Hindle, Green software engineering: the curse of methodology, PeerJ PrePrints 3 (2015) e1832.

[16] G. Pinto, F. Castor, Energy efficiency: a new concern for application software developers, Communications of the ACM 60 (12) (2017) 68–75.

[17] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, J. Clause, An empirical study of practitioners' perspectives on green software engineering, in: International Conference on Software Engineering (ICSE), 2016 IEEE/ACM 38th, IEEE, 2016, pp. 237–248.

[18] R. Pereira, Energyware engineering: Techniques and tools for green software development, Ph.D. thesis, Universidade do Minho (2018).

[19] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, J. Saraiva, Helping programmers improve the energy efficiency of source code, in: Proceedings of the 39th International Conference on Software Engineering Companion, ICSE-C '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 238–240. doi:10.1109/ICSE-C.2017.80.

[20] R. Pereira, Locating energy hotspots in source code, in: Proceedings of the 39th International Conference on Software Engineering Companion, IEEE Press, 2017, pp. 88–90.

[21] R. Abreu, P. Zoeteweij, A. J. C. v. Gemund, Spectrum-based multiple fault localization, in: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, IEEE Computer Society, 2009, pp. 88–99.

[22] R. Abreu, P. Zoeteweij, A. J. Van Gemund, On the accuracy of spectrum-based fault localization, in: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007), IEEE, 2007, pp. 89–98.

[23] A. E. Trefethen, J. Thiyagalingam, Energy-aware software: Challenges, opportunities and strategies, Journal of Computational Science 4 (6) (2013) 444–449.

[24] M. Couto, R. Pereira, F. Ribeiro, R. Rua, J. Saraiva, Towards a green ranking for programming languages, in: Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP 2017, ACM, New York, NY, USA, 2017, pp. 7:1–7:8, best Paper. doi:10.1145/3125374.3125382.
URL http://doi.acm.org/10.1145/3125374.3125382

[25] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, F. F. Rivera, D. S. Nikolopoulos, Power and energy implications of the number of threads used on the intel xeon phi, Annals of Multicore and GPU Programming 3 (1) (2015) 55–65.

[26] M. Kambadur, M. A. Kim, An experimental survey of energy management across the stack, in: ACM SIGPLAN Notices, Vol. 49, ACM, 2014, pp. 329–344.

[27] D. Li, S. Hao, W. G. Halfond, R. Govindan, Calculating source line level energy information for android applications, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM, 2013, pp. 78–89.

[28] S. Abdulsalam, Z. Zong, Q. Gu, M. Qiu, Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency, in: Proceedings of the 6th International Green and Sustainable Computing Conference, IEEE, 2015, pp. 1–8.

[29] R. Pereira, M. Couto, J. Saraiva, J. Cunha, J. P. Fernandes, The influence of the java collection framework on overall energy consumption, in: Proceedings of the 5th International Workshop on Green and Sustainable Software, GREENS '16, ACM, 2016, pp. 15–21.

[30] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. Saraiva, Energy efficiency across programming languages: How do energy, time, and memory relate?, in: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, ACM, New York, NY, USA, 2017, pp. 256–267. doi:10.1145/3136014.3136031.
URL http://doi.acm.org/10.1145/3136014.3136031

[31] M. Hähnel, B. Döbel, M. Völp, H. Härtig, Measuring energy consumption for short code paths using RAPL, SIGMETRICS Performance Evaluation Review 40 (3) (2012) 13–17.

[32] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, D. Rajwan, Power-management architecture of the intel microarchitecture code-named sandy bridge, IEEE Micro 32 (2) (2012) 20–27.

[33] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, K. Eder, Energy consumption analysis of programs based on xmos isa-level models, in: Logic-Based Program Synthesis and Transformation, Springer, 2013, pp. 72–90.

[34] A. Noureddine, R. Rouvoy, L. Seinturier, Monitoring energy hotspots in software, Automated Software Engineering (2015) 1–42.

[35] M. Dimitrov, C. Strickland, S.-W. Kim, K. Kumar, K. Doshi, Intel® power governor, `https://software.intel.com/en-us/articles/intel-power-governor`, accessed: 2017-10-12 (2015).

[36] L. S. Passos, R. Abreu, R. J. Rossetti, Spectrum-based fault localisation for multi-agent systems, in: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI'15), 2015, pp. 1134–1140.

[37] R. Real, J. M. Vargas, The probabilistic basis of jaccard's index of similarity, Systematic biology (1996) 380–385.

[38] P. E. Dombek, L. K. Johnson, S. T. Zimmerley, M. J. Sadowsky, Use of repetitive dna sequences and the pcr to differentiate escherichia coli isolates from human and animal sources, Applied and Environmental Microbiology 66 (6) (2000) 2572–2577.

[39] R. Rousseau, Jaccard similarity leads to the marczewski-steinhaus topology for information retrieval, Information processing & management 34 (1) (1998) 87–94.

[40] K. Liu, G. Pinto, Y. D. Liu, Data-oriented characterization of application-level energy optimization, in: Fundamental Approaches to Software Engineering, Springer, 2015, pp. 316–331.

[41] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, S. Tarkoma, Modeling, profiling, and debugging the energy consumption of mobile devices, ACM Comput. Surv. 48 (3) (2015) 39:1–39:40. doi:10.1145/2840723.
URL `http://doi.acm.org/10.1145/2840723`

[42] Monsoon, Monsoon solutions, inc., `http://www.msoon.com/LabEquipment/PowerMonitor/` (2018).

[43] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, K. Eder, Static analysis of energy consumption for llvm ir programs, in: Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES '15, ACM, 2015, pp. 12–21.

[44] N. Stulova, J. F. Morales, M. V. Hermenegildo, Reducing the overhead of assertion run-time checks via static analysis., in: PPDP, 2016, pp. 90–103.

[45] S. Hao, D. Li, W. G. J. Halfond, R. Govindan, Estimating mobile application energy consumption using program analysis, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, 2013, pp. 92–101.

[46] A. Pathak, Y. C. Hu, 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, ACM, 2012, pp. 29–42.

[47] S. A. Chowdhury, A. Hindle, Greenoracle: estimating software energy consumption with energy measurement corpora, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR, 2016, 2016, pp. 49–60.

[48] R. V. Hogg, E. A. Tanis, Probability and statistical inference, Vol. 993, Macmillan New York, 1977.

[49] A. Field, Discovering statistics using SPSS, Sage publications, 2009.

[50] R. Rosenthal, Meta-analytic procedures for social research, Vol. 6, Sage, 1991.

[51] R. Rosenthal, H. Cooper, L. Hedges, Parametric measures of effect size, The handbook of research synthesis (1994) 231–244.

[52] J. Cohen, Statistical power analysis for the behavioral sciences. 1988., Hillsdale, NJ: Lawrence Earlbaum Associates 2.

[53] G. Melfe, A. Fonseca, J. P. Fernandes, Helping developers write energy efficient haskell through a data-structure evaluation, in: 2018 IEEE/ACM 6th International Workshop on Green And Sustainable Software (GREENS), IEEE, 2018, pp. 9–15.

[54] T. D. Cook, D. T. Campbell, Quasi-experimentation: design & analysis issues for field settings, Houghton Mifflin, 1979.

[55] A. J. Ko, T. D. Latoza, M. M. Burnett, A practical guide to controlled experiments of software engineering tools with human participants, Empirical Software Engineering 20 (1) (2015) 110–141.

[56] R. Rosenthal, R. Rosnow, Essentials of Behavioral Research: Methods and Data Analysis, McGraw-Hill series in psychology, McGraw-Hill, 1984.
URL `https://books.google.pt/books?id=AftGAAAAMAAJ`

[57] R. R. Harmon, N. Auseklis, Sustainable it services: Assessing the impact of green computing practices, in: Management of Engineering & Technology, 2009. PICMET 2009. Portland International Conference on, IEEE, 2009, pp. 1707–1717.

[58] M. Cohen, H. S. Zhu, E. E. Senem, Y. D. Liu, Energy types, in: ACM SIGPLAN Notices, Vol. 47, ACM, 2012, pp. 831–850.

[59] C. Zhang, A. Hindle, D. M. German, The impact of user choice on energy consumption, IEEE software 31 (3) (2014) 69–75.

[60] C. Wilke, S. Richly, S. Gotz, C. Piechnick, U. Aßmann, Energy consumption and efficiency in mobile applications: A user feedback study, in: Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing, IEEE, 2013, pp. 134–141.

[61] D. Li, Y. Jin, C. Sahin, J. Clause, W. G. Halfond, Integrated energy-directed test suite optimization, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ACM, 2014, pp. 339–350.

[62] C. Sahin, F. Cayci, I. L. M. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, in: 1st International Workshop on Green and Sustainable Software (GREENS), 2012, IEEE, 2012, pp. 55–61.

[63] C. Bunse, S. Stiemer, On the energy consumption of design patterns, Softwaretechnik-Trends 33 (2) (2013) 1–2.
URL http://pi.informatik.uni-siegen.de/stt/33_2/01_Fachgruppenberichte/sre/01-BunseStiemer.pdf

[64] C. Bunse, H. Höpfner, S. Roychoudhury, E. Mansour, Choosing the "best" sorting algorithm for optimal energy consumption, in: Proceedings of the 4th International Conference on Software and Data Technologies, 2009, pp. 199–206.

[65] C. Bunse, H. Höpfner, E. Mansour, S. Roychoudhury, Exploring the energy consumption of data sorting algorithms in embedded and mobile environments, in: Mobile Data Management: Systems, Services and Middleware, 2009. MDM'09. Tenth International Conference on, IEEE, 2009, pp. 600–607.

[66] K. Rasmussen, A. Wilson, A. Hindle, Green mining: energy consumption of advertisement blocking methods, in: Proceedings of the 3rd International Workshop on Green and Sustainable Software, ACM, 2014, pp. 38–45.

[67] L. Cruz, R. Abreu, Performance-based guidelines for energy efficient mobile applications, in: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft '17, IEEE Press,

Piscataway, NJ, USA, 2017, pp. 46–57. doi:10.1109/MOBILESoft.2017.19.
URL https://doi.org/10.1109/MOBILESoft.2017.19

[68] C. Sahin, M. Wan, P. Tornquist, R. Mckenna, Z. Pearson, W. G. J. Halfond, J. Clause, How does code obfuscation impact energy usage?, Journal of Software: Evolution and Process 28 (7) (2016) 565–588. doi:10.1002/smr.1762.
URL https://doi.org/10.1002/smr.1762

[69] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, The impact of source code transformations on software power and energy consumption, Journal of Circuits, Systems, and Computers 11 (05) (2002) 477–502.

[70] J. J. Park, J.-E. Hong, S.-H. Lee, Investigation for software power consumption of code refactoring techniques., in: SEKE, 2014, pp. 717–722.

[71] I. Manotas, L. Pollock, J. Clause, Seeds: A software engineer's energy-optimization decision support framework, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 503–514.

[72] G. Pinto, K. Liu, F. Castor, Y. D. Liu, A comprehensive study on the energy efficiency of java's thread-safe collections, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016, 2016, pp. 20–31. doi:10.1109/ICSME.2016.34.
URL https://doi.org/10.1109/ICSME.2016.34

[73] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, A. Hindle, Energy profiles of java collections classes, in: Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 225–236.

[74] W. Oliveira, R. Oliveira, F. Castor, A study on the energy consumption of android app development approaches, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 42–52.

[75] S. Nakajima, Model-based power consumption analysis of smartphone applications., in: ACESMB@ MoDELS, 2013.

[76] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, G. M. Voelker, edoctor: Automatically diagnosing abnormal battery drain issues on smartphones., in: NSDI, Vol. 13, 2013, pp. 57–70.

[77] S. Nakajima, Everlasting challenges with the obj language family, in: Specification, Algebra, and Software, Springer, 2014, pp. 478–493.

[78] S. Nakajima, Model checking of energy consumption behavior, in: Complex Systems Design & Management Asia, Springer, 2015, pp. 3–14.

[79] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, S. Tarkoma, Carat: Collaborative energy diagnosis for mobile devices, in: Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems, SenSys '13, Roma, Italy, November 11-15, 2013, ACM, 2013, pp. 10:1–10:14.

[80] Y. Liu, C. Xu, S.-C. Cheung, J. Lü, Greendroid: Automated diagnosis of energy inefficiency for smartphone applications, IEEE Transactions on Software Engineering 40 (9) (2014) 911–940.

[81] A. Banerjee, A. Roychoudhury, Automated re-factoring of android apps to enhance energy-efficiency, in: International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM, IEEE, 2016, pp. 139–150.

[82] M. Couto, T. Carção, J. Cunha, J. P. Fernandes, J. Saraiva, Detecting anomalous energy consumption in android applications, in: Proceedings of the 18th Brazilian Symposium on Programming Languages, SBLP 2014, Springer International Publishing, 2014, pp. 77–91.

[83] R. Verdecchia, A. Guldner, Y. Becker, E. Kern, Code-level energy hotspot localization via naive spectrum based testing, in: H.-J. Bungartz, D. Kranzlmüller, V. Weinberg, J. Weismüller, V. Wohlgemuth (Eds.), Advances and New Trends in Environmental Informatics, Springer International Publishing, Cham, 2018, pp. 111–130.