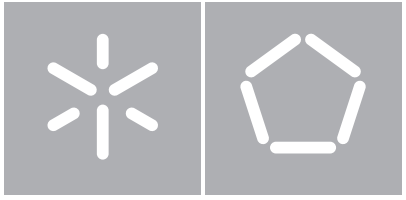
Universidade do Minho
Escola de Engenharia

Tiago Alves Carção

Spectrum-based Energy Leak Localization

Outubro 2014

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, within projects: FCOMP-01-0124-FEDER-020484, FCOMP-01-0124-FEDER-022701, and grant ref. BI2-2013_PTDC/EIA-CCO/116796/2010.



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Alves Carção

Spectrum-based Energy Leak Localization

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor Doutor João Saraiva
Professor Doutor Jácome Cunha

Outubro 2014

Acknowledgements

I want to thank both of my supervisors, Prof. João Saraiva and Prof. Jácome Cunha, due to their knowledge, dedication, experience, professionalism, innovative spirit, and their constant ability to discuss every detail throughout my Thesis, helped me greatly.

I also want to thank all of the members of the GreenLab @ UMinho where with the weekly meetings were able to provide useful contributions in this Thesis development.

To Joana, the person that is always on my side, that supported me in the hardest moments during this Thesis, always with an incentive word, and undoubtedly without I could not finish this path. Thank you so much.

To my laboratory buddies, Claudio and Rui that had spent some great time with me, in Romania, Italy, The Netherlands, Póvoa de Varzim and Australia.

To all of my friends, specially, David and Casimiro that always provide such a good and fun time when I am with them, and João and Daniel that are always a source of interesting and enriching discussions.

To my little brother that is not so little anymore, a big thanks for being a person that I can always count on and discuss the various subjects of life, politic and sports.

And finally, I would like to thank my parents that always supported me emotionally in my entire life, and gave me the opportunity to be who I am today. Thank you.

Abstract

For the past few years, we have begun to witness an exponential growth in the information and communication technologies (ICT) sector. While undoubtedly a milestone, all of this occurs at the expense of high energy costs needed to supply servers, data centers, and any use of computers. Associated with these high energy costs is the emission of greenhouse gases. These two issues have become major problems in society. The ICT sector contributes to 7% of the overall energy consumption, with 50% of the energy costs of an organization being attributed to the information technology (IT) departments.

Most of the measures taken to address the high level of energy consumption have been on the hardware side. Although is the hardware that does consume energy, it is the software that operates that hardware. As a consequence, the software is the main responsible for the energy consumed by the hardware, very much like a driver that drives/operates a car influences drastically the fuel consumed by the car.

This dissertation proposes and implements a methodology to analyze the software energy consumption. This methodology relates energy consumption to the source code of a software application, so that software developers are aware of the energy footprint that he/she is creating with his/her application. The proposed technique interprets abnormal energy consumption as software faults, and adapts a well-known technique for locating faults on programs's source code, to locate "energy faults", that we name as "energy leaks".

This methodology has been fully implemented in a software framework that monitors the energy consumed by a software program and identifies its energy leaks, given its source code. Moreover, a list of problematic parts of the code is produced, thus, helping software developers identifying energy faults on their source code. We validate our findings by showing that our methodology can automatically find energy leaks in programs for which such leaks are known.

With this results, one intends to provide help to the development phase and to generate more energy efficient programs that will have less energy costs associated with, while supporting practices that promote and contribute to sustainability.

Resumo

Localização de falhas de energia baseada no espectro do programa

Nos últimos anos, temos vindo a assistir a um crescimento exponencial no sector das tecnologias de comunicação e informação (TIC). Contudo, e apesar de, inquestionavelmente, se tratar um marco importante, tudo isto ocorre à custa de altos gastos de energia necessários para alimentar servidores, centros de dados e qualquer uso de computadores.

Paralelamente, associado aos altos custos de energia estão as emissões dos gases de efeito de estufa. Estas duas questões têm-se tornado grandes problemas da sociedade. O sector das TIC contribuí para 7% do consumo global de energia, o que representa, para o departamento de Tecnologias de Informação de uma organização, 50% de custos, associados, à energia.

A maioria das medidas adotadas para resolver o nível elevado do consumo de energia, têm sido feitas do lado do *hardware*. Apesar de ser o *hardware* que consume energia efectivamente, é o *software* que opera esse *hardware*. Como consequência deste facto, o *software* é o maior responsável pela energia consumida pelo *hardware*, tal como um condutor que dirige/opera um carro influencia drasticamente o consumo de combustível de um carro.

Esta dissertação propõe e implementa uma metodologia para analisar o consumo de energia por parte do *software*. Esta metodologia relaciona o consumo de energia com o código fonte de uma aplicação, permitindo que os desenvolvedores das aplicações estejam conscientes da pegada de energia que a sua aplicação está a ter. A técnica proposta interpreta um consumo de energia anormal como falhas no *software*, e adapta uma técnica de localização de falhas em código fonte bem conhecida, para localizar falhas de energia denominadas *energy leaks*.

A metodologia foi implementada numa *framework* que monitoriza a energia consumida por uma aplicação e dado o seu código fonte, identifica as suas falhas energéticas. Como adição, uma lista das partes problemáticas do código é produzida, ajudando assim os desenvolvedores a identificar as falhas de energia no seu código. Validamos os nossos resultados mostrando que a nossa metodologia consegue automaticamente encontrar falhas de energia em programas para os quais essas falhas são conhecidas.

Com estes resultados, pretende-se contribuir com uma ajuda na fase de desenvolvimento e na criação de programas mais eficientes a nível energético que terão menores custos de energia associados, ajudando a práticas que promovem e contribuem para a sustentabilidade.

Contents

Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

CPU Central Processing Unit

DAQ Data Acquisition

DRAM Dynamic Random Acess Memory

GPU Graphics Processing Unit

EPA Environmental Protection Agency

IT Information Technology

ICT Information and Communication Technologies

IDE Integrated Development Environment

JNI Java Native Interface

JVM Java Virtual Machine

MBD Model-based Diagnosis

MHS Minimum Hit-set

MVC Model-View-Controller

OS Operative System

RAPL Running Average Power Limit

SFL Spectrum-based Fault Localization

SI International System of Units

SPELL Spectrum-based Energy Leak Localization

List of Figures

List of Tables

Listings

1. Introduction

Currently, we are witnessing a technological era where information media has grown exponentially, with billions of users. Almost everyone has access to computers, and the internet is accessible virtually anywhere, which is undoubtedly a milestone in the field of content delivery [Guelzim and Obaidat, 2013].

The problem with this globalization is that all of this occurs at the expense of energy consumption that is the necessary and indispensable element to supply servers, data centers and any use of computers [Guelzim and Obaidat, 2013]. The energy required to meet the growing demand for power to run the Information and Communication Technologies (ICT) infrastructure and storage its information, grows faster along with the widespread diffusion of cloud services over the internet [Ricciardi et al., 2013]. This fast and growing power consumption attracted the attention of governments, industry and academia [Zhang and Ansari, 2013]. Also, associated with this energy consumption, is the emission of greenhouse gases. These two issues are becoming a major problem in the society of information and communication [Ricciardi et al., 2013].

The costs of energy consumption in the field of ICT will be increasing over the next 20 years [Rühl et al., 2012] which alone is a great incentive for green practices. The ICT with its intrinsic properties and with their use, helps to reduce the energy consumption in other sectors. Nonetheless, it has a forecast increase in their energy consumption. Its share of 7% in global energy consumption will be increased to more than 14.5% in 2020 [Vereecken et al., 2010].

Recently, the world has witnessed an exponential growth of IT devices. Data centers are nowadays a common term in the vocabulary of informatics and all the big technological companies have this kind of infrastructure. Although these infrastructures endure what is widely known as the cloud, and upon all the benefits that this feature brings, maintaining data centers carries substantial energy costs related to supply (huge set of machines and devices as well as cooling systems) [Mouftah and Kantarci, 2013]. Adding up to the costs, there is still a large amount of greenhouse gases in this eco-system. With what is expected to be a future reality in a very short period of time, the Internet of Things [Atzori et al., 2010],

it is expected that the network of devices present increases significantly. That fact itself will imply that there is an infrastructure capable of handling this increase of information which will naturally result in a boost of global energy consumption.

The energy consumption has an immediate impact on the business value. In fact, the energy costs associated with information technology departments constitute approximately 50% of the overall energy costs of the entire organizations [Harmon and Auseklis, 2009]. There is also electricity that is wasted, and potentially avoidable, that is leading to high operating costs [Zhang and Ansari, 2013]. Thus, this raises the need and expectation of reducing the energy costs and the impact on the environment, by directing attention to these issues [Harmon and Auseklis, 2009].

Energy efficiency requires a thorough investigation to discover and understand everything that is related with it [Zhang and Ansari, 2013]. ICT services require the integration of sustainable practices for green computing to meet sustainability requirements [Harmon and Auseklis, 2009]. This term, **green computing**, refers to the practice of using computing resources more efficiently, maintaining or increasing their overall performance. Although the original conceptual already exists for two decades now, only since the last decade has it received more attention [Harmon and Auseklis, 2009].

Thus, **green computing** paradigms are emerging to reduce energy consumption, the resulting emissions of greenhouse gases, and operating costs [Ricciardi et al., 2013], that is, researchers and companies are trying to find solutions that make all these systems energy efficient [Mouftah and Kantarci, 2013].

The industry is becoming more active in the area of green computing, increasingly attempting to reduce costs and energy consumption. For example, Symantec, using the monitoring of their resources, found there were some resources that were being squandered and by implementing measures to reduce this waste they saved close to $2 million and more than 6 million kilowatts of energy [Symantec, 2008a,b]. Google also made some changes, using customized cooling systems in their data centers bettered the energy consumption values [Google, 2014].

New researches and discussions are being addressed to enable new solutions that use energy as an additional constraint, minimizing its consumption [Ricciardi et al., 2013].

This Thesis addresses in detail green computing in the energy consumption of software. Nowadays when one says that a program is efficient, the term efficient encapsulates the notion that software is fast to execute and performs the task without requiring a lot of resources (CPU, memory, disk, etc.). However, efficiency can also be applied to energy, and it is exactly this notion that one needs to change in the consciousness of the programmer, the notion that it is also possible to have an efficient software in terms of energy.

All the hardware components of ICT consume a constant power consumption to be running. When they are performing operations they increase this power consumption. These operations are directly related to the software needs, which makes the study of energy consumption quite pertinent in software. Due to the fact that it is software that makes hardware perform its tasks, up to 90% of the energy used by ICT can be attributed to software applications running on them [Standard, 2013]. The design of software has significant impact on the amount of energy used [Standard, 2013]. So it is very important that software engineers are aware of the consumed energy by the software they design, in order to project more efficiently in regards to energy consumption, knowing precisely where the high consumption parts are and how to correct them.

1.1. Research Questions

During my Thesis work, three important questions arose, relative to the design of a technique to analyze the software and identify its energy leaks (anomalous values of energy consumption). These questions when answered, help to better understand what was made, and if/how we were able to accomplish its challenges.

1. *Can we define a methodology to analyze the energy consumption of software source code?*

2. *Is it possible to adapt a general purpose fault localization algorithm to the context of energy leak localization?*

3. *Can we find energy leaks in software source code?*

By the time I conclude my dissertation, I plan to easily answer all of these questions.

1.2. The Solution

The objective of this Thesis is to create a technique to analyze a program's execution with a test suite and discover the energy leaks present in the software program.

There are two points of views in the energy consumption: the energy consumption and power consumption. While the energy consumption is the total energy consumed during a period of time and is measured in joules (J), defined in the International System of Units (SI), the power consumption is the energy consumed per unit time (J/s), or as is defined in the SI, watts (W). The energy consumption indicates, for a given component, the total energy consumed which is the desired information when we want to extract information from an analysis on where we can make some changes that improve instantly the energy performance of the program's execution, the choice taken in this Thesis. On the other hand, the use of power consumption is useful when we want to find the software components that consume more amount of energy per time and therefore their utilization in multiple programs against less consuming components is discouraged. This type of information can be used to extract energy consumption patterns, which is set for future work.

With the Thesis objective in mind a methodology to accomplish this goal was defined. This methodology has three different phases. For each of these phases, a sub-technique was developed. In the first phase the software code is modified to also extract the execution information of each program's constituent besides execute it. This information is structured and represents the constituent energy consumption, the time of its execution and the number of times it was used. After this process the software is compiled and ran with a test suite. In the second phase, the execution data produced by the program's execution are collected, aggregated and treated and the information is then passed into the final phase. In the final phase and using a technique based in the program's spectrum and its execution data, the data is evaluated and the information about which are the energy leaks is obtained. This methodology in conjunction with the three phases defined, accomplishes the objective previously set.

1.3. Structure of the Dissertation

This dissertation is organized as follows:

Chapter 2 - Green Computing and Software Fault Localization Techniques - contains the State of the Art, with information on green computing evolution and the emerging area of green software computing and introduces the software fault localization techniques establishing the relation between software energy leaks and software faults.

Chapter 3 - Instrumentation, Compilation and Execution - presents the methodology proposed, detailing each phase of the methodology.

Chapter 4 - The SPELL Framework - describes and showcases the framework developed, which contains and describes in more detail all the techniques and implementations presented in Chapter 3.

Chapter 5 - Validation - contains the process of validation of the methodology proposed.

Chapter 6 - Conclusion - concludes this dissertation with comments on the work done, results, and future work, along with answers to our research questions.

2. Green Computing and Software Fault Localization Techniques

2.1. Green Computing

The concept of green computing despite being a hot topic is a relatively old concept. It has emerged around the 90s when the awareness of the energy that was being used by IT devices was raised, which led the IT community to take some measures. One of the first measures taken under green computing was the assignment of a "certificate" to products that had a concern in terms of energy consumption, minimizing it while maximizing efficiency. This certificate (Figure 1) was applied to different peripherals, computers, monitors, printers, etc. One of the first real results of this awareness was the appearance of the stand-by functionality in the devices that made them entering in sleep mode after a period of inactivity.

Figure 1: *Energy Star certification symbol*

Despite the fact that this awareness already started 20 years ago, only just more recently, in the last decade, started to exist a more active concern with the reduction of energy usage.

With the predictions of an increase of the global energy consumption, countless associations begun to focus their attention on this issue. A number of organizations, including the USA's Environmental Protection Agency (EPA), have identified a number of processes, optimizations and energy alternatives in data centers and even in home appliances [Fanara et al., 2009]. Google was another of the organizations that included in its research the topic of green computing and has already achieved some results, reducing its data centers' energy consumption [Google, 2014].

Another aspect of IT is the use of personal computers, and recently (and exponentially growing) smartphones and tablets. These devices have an intrinsic concern for energy usage since their power supply is taken from a battery which has a limited capacity. The less energy consuming components of these devices are, the less power will be consumed, and therefore it is possible to use these devices during a longer period of time. So, with this in mind, all the companies involved with these devices have a great interest in this field.

Energy wise, version after version, Intel, the largest producer of processors for computers, smartphones, tablets, etc., has had a concern in obtaining maximum efficiency while lowering the power consumption of its processors. This development has permitted after each release, on the one hand to reduce the energy consumption in the use of processors, and on the other hand to extend the usage time of portable battery powered devices [Ralph, 2011; Crisostomo, 2012; Anthony, 2013].

The interest in this area exists and has strong promoters which is already remarkable. However, one can not ignore the fact that the ITs consist of two artifacts of different types: hardware and software. If on one side a lot has been done in order to decrease the power consumption of the hardware, as already shown – which is understandable since hardware changes do not alter the normal functioning of the software and allow immediate energy savings to be made –, either by physical limitations or because more needs to be done to reduce the energy usage, software is an obvious target.

2.1.1. Green Software Computing

The concern for energy usage in software has already started to happen although on a smaller scale when compared to the hardware, and has already been dubbed the *Green Software Computing*.

Slowly we start to see some initiatives from companies that support some of the world's major operating systems (OS) such as Apple's Mac OS X and iOS, and Google with Android. Indeed Apple, in its most recent versions of the operating system for desktop (Mac OS X), by using only the operating system software, was able to improve the energy performance of their computers, thus allowing the battery life to be extended, in some cases, up to 4 hours [Brownlee, 2013]. Regarding the mobile OS, iOS and Android devices already have

tools that allow the user to check the battery consumption profile of applications. Apple already allows its developers in its integrated development environment (IDE) – Xcode – to make an energy profiling to their applications. Android in its new version (Lolipop) is scheduled to receive energy profiling tools aimed at software engineers.

A recent study showed that software developers are aware and interested in the green software domain [Pinto et al., 2014]. This study demonstrated that there is an interest in the community to learn more about this area and try to find out what may be the causes of high energy consumption and possible ways to address them. Also note that software engineers feel there is a lack of tools to support this identification process and optimization.

To make greener software, besides requiring the energy consumption values, developers also need to know what zones of code are hotspots, or areas where the power consumption is excessive. These areas can be seen as *red* zones and must be the first ones to be investigated.

In order to proceed with the identification of these red areas, one needs to be able to measure the energy consumption. As mentioned, research in green software computing is still in an early stage and therefore the techniques and tools that exist to measure this consumption are incomplete and insufficient. To overcome this fact in some cases estimates are used but some of these estimates are not reliable and are not precise [Hurni et al., 2011]. Although external devices can be used, they will only allow to measure the total power consumption for a period of time. This option may have read errors that are always associated with the reading of values in external devices. Adding to these difficulties, there is also the fact that the measurement of consumption is done on the whole system and not only on the desired applications.

Intel, as a manufacturer of processors, and also as a promoter of green computing, since 2012 began to worry about providing tools to software engineers to gain access to energy consumption by existent on-chip components (either the processor, DRAM or even on-chip GPU). This tool is provided as an API and is named Running Average Power Limit (RAPL) [Rotem et al., 2012]. RAPL is an interface that allows system calls to consult the values of energy consumed by each hardware component. These intakes are updated by the processor that will from time to time update some special registers in memory reserved for this purpose. Thus, by reading these registers one can know the recorded energy

consumption. There are studies that prove that the measurements made by this interface are accurate and trustworthy [Hähnel et al., 2012]. However, RAPL only reports on-chip consumption leaving aside peripherals such as the hard drive, and non-integrated GPUs and motherboard.

To address this lack of information, academia started to construct their own tools to allow them to monitor the power consumption.

As previously said, the tools to run energy profiling are short in number and often lack the desired accuracy. Because of this, several institutions have developed their own methods for monitoring power consumption. Software Improvement Group (SIG), a company that is linked to qualitative analysis of software, in collaboration with Amsterdam University, developed a laboratory related to energy. This laboratory has developed a piece of hardware that can be connected to any computer hardware component and also connected to a device termed Data Acquisition (DAQ) that will produce as output the power consumption of the components connected to it (Figure 2) [Ferreira et al., 2013]. At SIG they also already researched the efficiency of e-services energy and proposed some indicators to improve its consumption [Arnoldus et al., 2013]. They also defined some metrics to quantify how the values of the optimum of system-relative energy efficiency and its utilization are aligned. This quantification also allows the comparison of two distinct services [Grosskop and Visser, 2013; Grosskop, 2013].

Li et al. [2014] also developed a similar technique but for mobile devices. One can also use hybrid variants for measuring the power consumption: Li et al. [2013] showed that combining hardware analysis based on power measurements, and software statistical modelling, at least in Android, is possible to calculate values of the source line energy consumption.

To measure the energy, which can be done using external or internal devices, and with a higher/lower level of refinement, some contributions have already been made.

Using a model-based policy, Zhang et al. [2010] during hers PhD developed an application for Android that allows any application's energy consumption to be monitored. The limitations of this application are largely associated with the problems of using models, i.e., the need to calibrate the model for the environment where the application is running. [Couto et al., 2014; Couto, 2014] attempts to solve this and other limitations by creating a

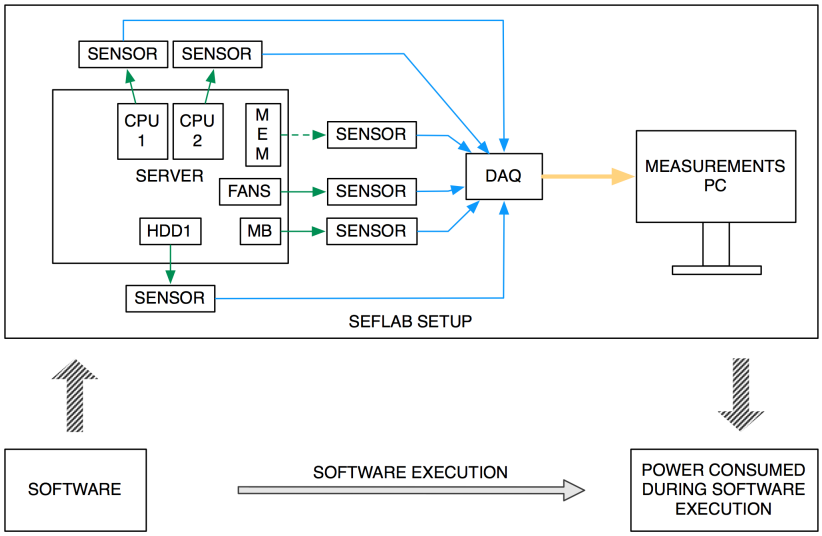


Figure 2: *SEFLab infrastructure [Ferreira et al., 2013]*

dynamically calibration of the models for any smartphone.

Hönig et al. [2013] published a technique that uses a model-based technique to generate information about software energy consumption. This technique, illustrated in Figure 3, uses symbolic execution and execution knowledge stored in a database, to predict energy consumption of a particular program.

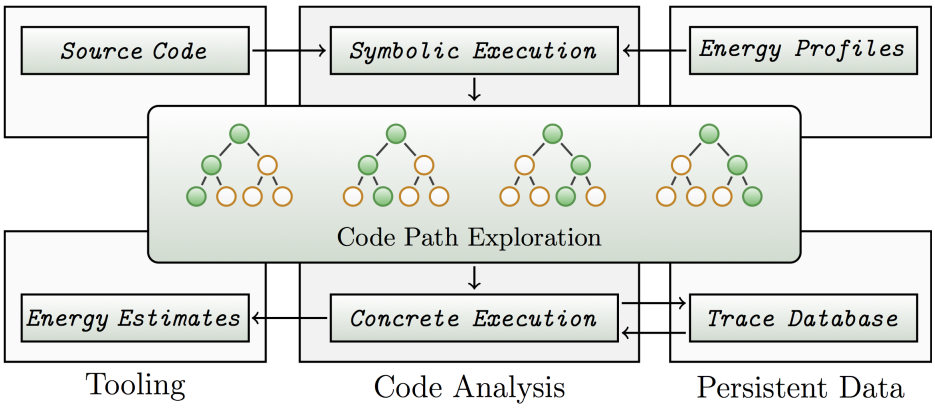


Figure 3: *SEEP technique that tries to bring energetic advices to the development process [Hönig et al., 2013]*

Also, in an attempt to provide energy information for a particular program, Noureddine et al. [2014] developed a technique for the instrumentation and collection of the energy

usage data in Java (*JalenUnit*) where they analyze the power consumption of a method by varying the method's data input, and validate the results by inspecting the code manually and confirming that its implementation lead to such results.

One of the current practices in the development of applications is, before publishing them to the public, run an obfuscation tool on the source code trying to deter copying of software logic. Using this as motivation, Sahin et al. [2014] investigated and demonstrated that obfuscation has a significant statistical impact and is more likely to increase the energy usage. These conclusions are an indicator that the way the code is written has an influence on energy consumption.

Gutiérrez et al. [2014] did an energy consumption study in multiple Java Collections. They produced as results what were the collections that had higher intakes of energy or that were more energetically efficient. In conjunction with this analysis, they also developed a framework which taking into account the data obtained, refactors the Java source code to use the collections that statistically consume less energy.

A common practice in the software world, the use of patterns, was also questioned at the energy level. Vásquez et al. [2014] presented a qualitative and quantitative study of the high energy consumption in API calls and patterns used in Android. Their findings indicate that there are patterns that have a significant impact on the energy consumption, such as the Model-View-Controller pattern. Sahin et al. [2012] also did an analysis of the impact on the energy usage in software design patterns and concluded that each design impacts but not in a similar way, the energy consumption values.

Gonçalves et al. [2014] did a study about how a Database Management System can use some energy consumption indicators to build query plans and obtain a SQL query that consumes less energy. The results that they obtained suggested that this approach could be successful to produce query plans that consume less energy.

2.2. Software Fault Localization

It is becoming more common to have IDEs offering tools to provide the values of power consumption of the programs being written. Although the values of energy consumption

are already provided to developers, the notion of what they mean and what relevance the consumption of certain software components have in the program's consumption is yet to be determined.

In this dissertation, we propose a set of techniques and tools to determine *red* areas in the software energy consumption. In this context, a parallel is made between the detection of anomalies in energy consumption in software during execution of the program and the detection of faults in the execution of a program. Establishing this parallelism, we will adapt fault detection techniques, used to investigate the failures in the execution of a program, to be used in the analysis of energy consumption.

When it comes to identify faults in programs there are two main possibilities: reasoning approaches (i.e. Mayer and Stumptner [2008]) or statistical (i.e. Zheng et al. [2003]). The reasoning approaches to fault localization build a model of the correct behaviour of the system using prior knowledge which allow to extract accurate conclusions about failures that may be happening. However, and because in a model we have to define the complete system, when applied to energy, at least for now, it would be impractical to obtain an energy model of the software. It would be necessary to take into account the system settings and all the implications that a change in the model would lead to, energetically wise. The statistical analysis, based on the implementation of the program using the source code, does not allow taking totally accurate conclusions, but allows useful information to be extracted with relative ease.

So, knowing the two main options, the statistical analysis technique of fault localization is the most appropriate. Since its foundations rely on an analysis of the program based on its implementation (in its source code), one does not need to model the entire system. Within the statistical analysis techniques for locating faults, the technique of using the program spectrum is more efficient than the use of dynamic slicing [Korel and Laski, 1988] and therefore the technique that stands out as candidate, with very good results in this field, is the Spectrum-based based Fault Localization technique (SFL) [Abreu et al., 2009].

2.2.1. Spectrum-based Fault Localization

A program spectrum is a set of run-time execution data of a program [Reps et al., 1997]. There are different types of program spectro that can be used [Harrold et al., 2000], Table 1 shows some examples. To better understand those types of program spectro, let us consider the use of the block-hit type, in the Listing 1 (it calculates the largest of three numbers). One can see what is actually considered as a block-hit in the program execution. The spectrum of a block-hit program is a set of flags that will reflect if the condition of the block is used or not.

Table 1: *Types of program spectrum [Harrold et al., 2000; Abreu, 2009]*

Name	Description
Statment-hit	statements that were executed
Block-hit	conditional branches that were executed
Path-hit	intraprocedural, path that was executed
Output	output that was produced
Time spectra	execution time of program's components

In SFL, the hit spectrum is used to build a matrix A, $n \times m$, where the m columns represent the different parts of the program during n executions (independent, i.e. the result of each execution does not influence the next) as can be seen in Figure 4 (left-hand part). In this hit spectrum (a_{nm}), the value 0 means that part m was not executed in execution n, and the value 1 means it was. The SFL representation also presents one column vector e corresponding to the errors (right hand part of Figure 4). Each element of this vector represents the presence of an error in the result of the test, where the value 0 means that no error occurred and 1 otherwise. The objective of spectrum-based fault localization technique is to try to find which components of the program are more likely to being faulty by using their column representation and discovering which component column best explains the existence of the errors represented in the vector of errors. This similarity of vectors is quantified by coefficients of similarity [Jain and Dubes, 1988]. The existing test vector can be obtained in different ways. In SFL, there is the notion of an oracle that enables the vector error to be generated with the consultation of the oracle state. This oracle, in the case of detecting faults in a program, can be seen as the supposed output that the program

Listing 1: *Instrumented program to the block level*

```
int largestNumberAmongThreeNumbers(int a, int b, int c) {
    int res;

    if (a > b) {
    //block (c1)
        if (a > c) {
            //block (c2)
            res = a;
        }
        else {
            //bock (c3)
            res = b;
        }
    }
    else {
    //block (c4)
        if (b > c) {
            //block (c5)
            res = b;
        }
        else {
            //block (c6)
            res = c;
        }
    }

    return res;
}
```

may have.

Given the coefficients of similarity existing in SFL techniques the best performing coefficient is the Ochiai [Abreu et al., 2006]

$$S_O = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \cdot (n_{11}(j) + n_{10}(j))}} \tag{1}$$

where $n_{11}(j)$ is the number of failed runs in which component j was involved, $n_{10}(j)$ is the number of successful runs where component j was involved, $n_{01}(j)$ is the number of failed runs where component j was not involved, and $n_{00}(j)$ is the number of successful runs where

$$
\begin{array}{c} \\ n \text{ spectra} \end{array}
\begin{array}{c}
\overset{m \text{ components}}{} \\
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1m} \\
a_{21} & a_{22} & \cdots & a_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nm}
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
\text{error} \\ \text{detection} \\
\begin{bmatrix}
e_1 \\ e_2 \\ \vdots \\ e_n
\end{bmatrix}
\end{array}
$$

Figure 4: *The spectrum-based fault localization model (A,e)*

component j was not involved.

In the case of coefficients of similarity, a value closer to 1 means that this vector is more likely to explain the result of the vector of errors. To better understand the spectrum-based fault localization technique we will use an example. Figure 5 presents the values of the Ochiai coefficients calculated for each m column vector, of applying the SFL technique to the program shown in Listing 1 with the inputs $\langle 2, 4, 1\rangle$, $\langle 5, 3, 1\rangle$, $\langle 5, 2, 7\rangle$, $\langle 3, 9, 12\rangle$, $\langle 1, 3, 1\rangle$ and $\langle 2, 1, 4\rangle$, and with the outputs $4, 5, 7, 12, 3$ and 4 respectively.

	c_1	c_2	c_3	c_4	c_5	c_6	e
	0	0	0	1	1	0	0
	1	1	0	0	0	0	0
	1	0	1	0	0	0	1
	0	0	0	1	0	1	0
	0	0	0	1	1	0	0
	1	0	1	0	0	0	1
$n_{11}(j)$	2	0	2	0	0	0	
$n_{10}(j)$	1	1	0	3	2	1	
$n_{01}(j)$	0	2	0	2	2	2	
$s_O(j)$	0.82	0.0	1.0	0.0	0.0	0.0	

Figure 5: *Result of SFL technique applied to Listing 1, indicating c_3 as the faulty component*

The last row of Figure 5 indicates that the component 3 (c_3) has the highest probability of being faulty, and the component 1 (c_1) is the closest second. In fact, if we consult the program in Listing 1 we can see that the block c_3 has an error, because it does not compare the value of b with c, which will lead to failure for some inputs. The fact that c_1 has such a hight probability can be explained because this component enclosures the faulty component and so, will also fail for some inputs.

3. Spectrum-based Energy Leak Localization Analysis

The process of energy analysis is dependent on the approach that one wants to define. The aim of this thesis is to conduct an energy consumption analysis of the software source code, so this process will focus on the source code level. The process takes as input a program yet to be compiled and a set of program tests, and provides information about the program's energy consumption.

What is proposed here can be seen as a generic methodology to be followed for the energy usage analysis on an application's source code. The method is generic and therefore can be applied to any language/programming paradigm.

The basis of this methodology is the methodology used in the SFL, i.e., we have a program and we want to extract its spectro in different tests in order to draw conclusions. Because this is an energy consumption analysis the data collected must be more informative about the program's execution. This is why, at the end of the executions where it will exist the execution data non-structured, this data should be structured hierarchically so one can analyze it. After having this execution information, as in SFL technique, it is analyzed and conclusions are extracted.

In the proposed methodology, one can identify three distinct steps:

1. Instrumentation, compilation and execution
2. Execution information processing
3. Energy data analysis.

Over the following sections, each step will be explored in detail.

3.1. Instrumentation, Compilation and Execution

As seen in Section 2.2.1, when one wants to identify the spectrum of a program implementation it must specify the level on where the analysis will be performed. Depending on the programming language where the target program was developed, this granularity may vary. For instance, in the C language, one can have Libraries > Files > Functions > Block

of code > Line of code. In a second example, Java, one can have Packages > Classes > Methods > Block of code > Line of code. In other languages/paradigms there may exist other components. Consequently, for each language it will always be necessary to define the desired granularity.

After having defined the level of source code for which one wants to retrieve the information, it is also necessary to define the desired information to collect with the instrumentation. The final goal of the process is to analyze the program's energy consumption. Therefore, the logic data to be gathered is the information related with the energy consumption of the computer hardware components. As examples of hardware components there are the CPU, CPU cache, DRAM, hard drive disk, fans, graphics card, motherboard, and other machine specific peripherals, and the program specific components (for example, the use of the mouse in a specific program). To complement this process, there is information that can be useful to retrieve conclusions about the profiling of energy: execution time, CPU frequency, CPU temperature, etc.

3.1.1. Instrumentation

With the level of source code granularity and the information to collect chosen the next step is to perform the instrumentation itself. To do so one can start by write by hand on the source code the instructions to collect the data after the execution, but this is an inefficient, time consuming and not scalable process. So, in order to obtain an automated instrumentation, a structure that represents the program and can be modified to contain the collecting instructions must be defined. The use of such structure is a technique that modern compilers already use in their compiling processes and is called Abstract Syntax Tree (AST). The AST represents the constituents of a program in a hierarchical manner. For instance, in Figure 6, we have the abstract syntax tree of the largest of three numbers program, presented in Listing 1. This structure allows changes by using operations without having concerns about the syntax structure of the source code file. This operations can be: add, remove or update nodes. Currently there are front-ends for almost every language, that offer a parser with the AST construction. Therefore, the instrumentation becomes simpler for almost every language.

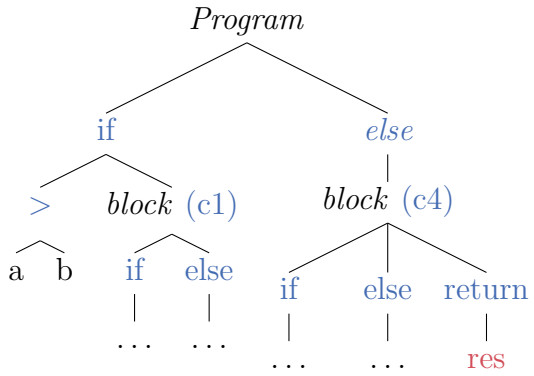


Figure 6: *The abstract syntax tree of a program (the largest of three numbers program, presented in Listing 1)*

Besides having to collect spectrum information from the program, one also needs to collect energy information, and therefore one needs to define how it is obtained. The source of this information may vary. It can be an external device that measures the overall energy consumption, a set of system calls that allow greater precision, or even a pre-defined model. To the instrumentation here defined it is assumed that there is a framework that allows to accurately measure the power consumption within a certain range (depending on the granularity level). So, to define this range, information nodes to read the context information and print it to the standard output are both added before and after the granularity level content. An example of an AST of the largest of three number program, shown in Listing 1, instrumented can be seen in Figure 7.

The syntax of this information generated from all components has to be produced in a strict format because it will serve as input to the next phase (Section 3.2).

We want to systematize the process of analysing software in terms of energy, and to do so, we will create an activity diagram that will be built throughout this dissertation. Figure 8 shows the start of this activity diagram, with the process mentioned above.

3.1.2. Compilation and Execution

After the instrumentation is made in the AST, the software source code has to be compiled, but now containing the needed instructions to collect the energy usage. After the

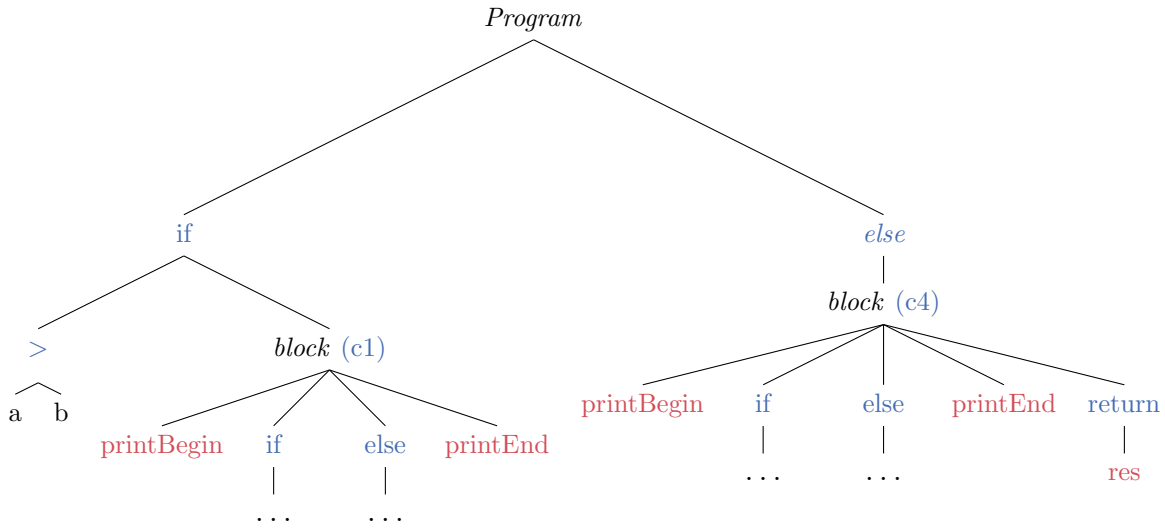


Figure 7: *AST of the largest of three number program instrumented at block level with nodes to extract energy information*

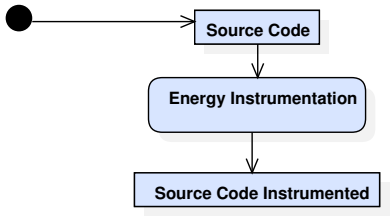


Figure 8: *Process of instrumentation*

compilation, the compiled program must be ran with a set of different inputs (test suite), that will test the program code. The more diverse and complete in terms of coverage of the program these tests are, the better analysis of the information extracted from the software implementation can be made [Cai and Lyu, 2005].

Continuing to build on the methodology and process defined in Figure 8, Figure 9 adds this step to the activity diagram. The resultant execution data will serve as input to the next phase as already mentioned. The general execution of this first phase is shown in Algorithm 1.

The input to the next phase (described in the following Section) is the output from the instrumented program execution. This output is written in a flat and sequential way, representing the order that the components were used in the program. Listing 2 shows a sample of the output of an execution with 3 tests of largest of three number program

Algorithm 1 Program Instrumentation, compilation, and execution with a test suite

```
1:  procedure INSTRUMENTATE, COMPILE, AND EXECUTE TESTS
2:
3:      for all module in software do
4:          moduleInstrumented ← energyInstrumentation(module)
5:          softwareInstrumented ← softwareInstrumented + moduleInstrumented
6:
7:      softwareCompiled ← compile(softwareInstrumented)
8:
9:      for all testCase in testSuite do
10:         output ← execute(softwareCompiled)
11:         instrumentationOutput ← instrumentationOutput + output
12:
13:     return instrumentationOutput.
```

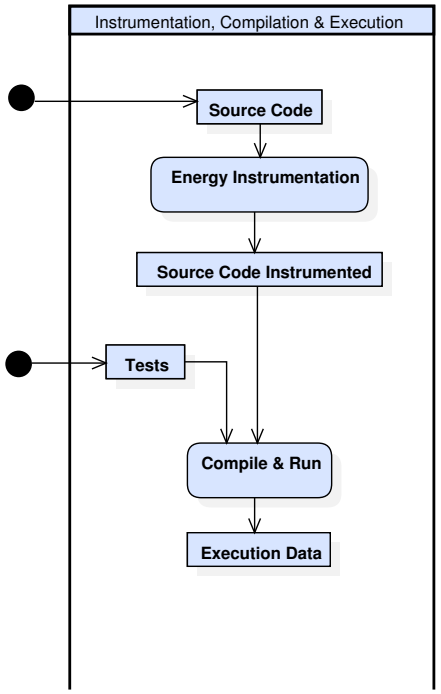


Figure 9: *Process of instrumentation, compilation and execution of the software with the test suite*

previously instrumented. Each line represents or the beginning of component execution, or the end of its execution and has inside its square brackets the execution information. This output follows the syntax grammar defined in Appendix A. In the next Section we will explain in detail the format of this output.

So, whatever is the language of the program instrumented, the output form will have the same syntax for all languages and paradigms, which makes the following phase independent of any programming language or paradigm.

Listing 2: *Example of an ouput of a execution of the largest of three number instrumented program ran with 3 tests.*

```
/*test 1*/
> c1 [ time = 0,  cpu = 32,  dram = 7  ]
> c2 [ time = 0,  cpu = 65,  dram = 12 ]
< c2 [ time = 7,  cpu = 120, dram = 16 ]
< c1 [ time = 15, cpu = 140, dram = 19 ]

/*test 2*/
> c4 [ time = 0,  cpu = 34,  dram = 8  ]
> c5 [ time = 0,  cpu = 64,  dram = 14 ]
< c5 [ time = 6,  cpu = 121, dram = 17 ]
< c4 [ time = 13, cpu = 130, dram = 20 ]

/*test 3*/
> c4 [ time = 0,  cpu = 31,  dram = 5  ]
> c6 [ time = 0,  cpu = 64,  dram = 7  ]
< c6 [ time = 6,  cpu = 117, dram = 12 ]
< c4 [ time = 14, cpu = 134, dram = 14 ]
```

3.1.3. Process Instantiation

The process of instrumentation, compilation and execution was instantiated to instrument programs written in C language, which, by doing so, allows the analysis of the energy consumption of C programs. We choose the C language because it is well established in the community and provides access to a good number of repositories of robust open source software that can be tested energy wise.

To do this instrumentation it was necessary to find an instrumentation tool that allowed the extraction of the C language AST from a program. The natural language of choice was a fairly complete tool that actually serves as a C front-end for the LLVM compiler and that among the many features already available, can build the program's AST from a file; it is

called the Clang[1] framework.

The accuracy chosen for the analysis and consequently to instrument the code was defined at the function level, which means that to retrieve the information required for the analysis one needs to know where is the beginning and the end of each function. The granularity choice is related with the precision one wants to extract and analyze information and in this case it is also limited to the existing tools and their accuracy.

The chosen framework to collect the energy data required for posterior instrumentation and analysis was the Intel Power Gadget framework[2]. This framework works based on the framework RAPL and provides information on energy consumption and performance of the CPU. To measure the execution time it is used the Time library in C[3].

It was developed a small program in C++ linked with Clang that allows to build the AST, add the nodes and regenerate the source code. At the end of the instrumentation, one generates again the program's source code, as shown in Listing 3, and compile it.

Listing 3: *Generic instrumented C program with information to log energy consumption*

```
void function() {
    startMeasuring (Regist information, Display begin)

    /* PROGRAM EXECUTION BEHAVIOR */

    endMeasuring (Display end & information)
}
```

The compiled program is then executed with a test suite, and for each test, it produces the information about its energy consumption.

The Section 4.1 provides more details about this phase implementation.

[1] http://clang.llvm.org/
[2] https://software.intel.com/en-us/articles/intel-power-gadget-20
[3] http://www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_19.html

3.1.4. Instrumentation Case Study: GraphViz

As an initial proof of concept and to apply the instrumentation to a robust application fully established, it was decided to choose a tool that had heavy processes and did some intensive processing to generate its output. The chosen tool was an open-source tool called GraphViz [Gansner and North, 2000]. GraphViz is a software package that enables the design of graphs, processing and generating the corresponding view. The instrumentation tool (explained in Section 4.1) was applied to the software package with about 18 tests. The OS of the computer where these tests were run was the MacOS X 10.9. These tests ran GraphViz with different generation flags and different input graphs. In Figures 10 and 11 we show the results (for the sake of visualization, some functions, modules and tests are omitted). Figure 10 shows the energy consumption (in milijoules) (y-axis), of GraphViz functions (x-axis) – represented with numbers to simplify its visualization –, for the different tests cases – represented with different colors. Figure 11 shows the energy consumption (in milijoules) (y-axis), of GraphViz modules (x-axis), for the different tests cases – represented with different colors.

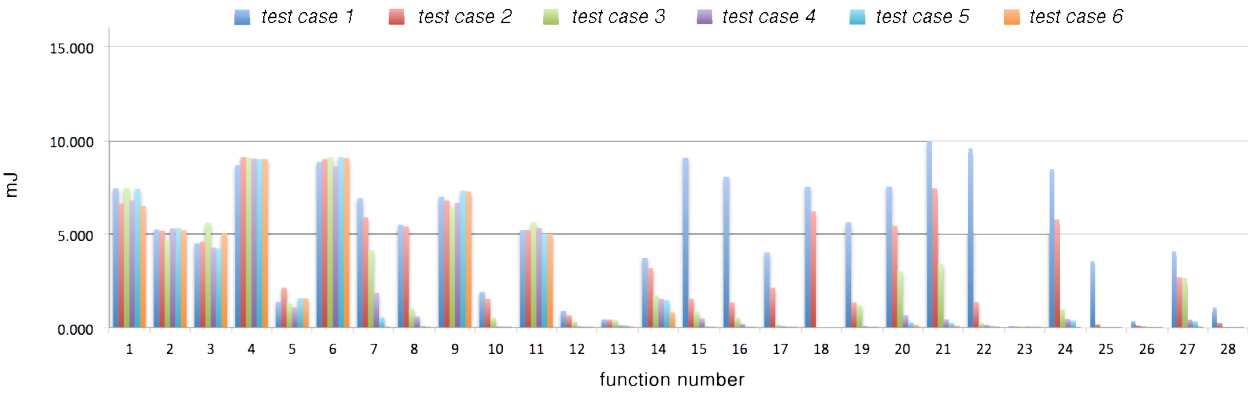


Figure 10: *Energy consumption of GraphViz functions*

These graphs show that different inputs and different flags have different energy consumption values which by itself is and indicator that an analysis can be made with different tests to extract energy usage information. This was one of the first results that motivated further research.

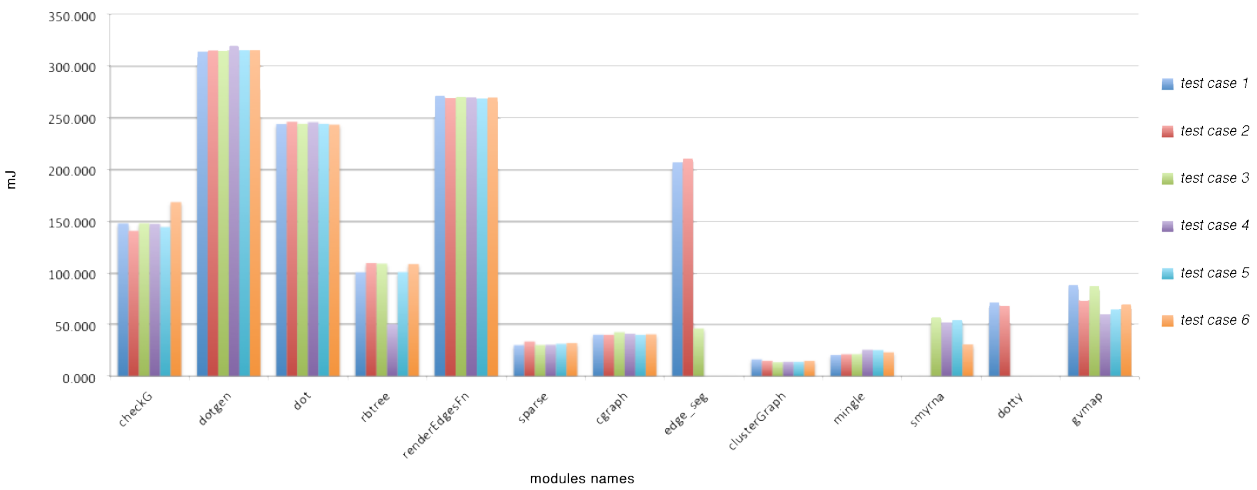


Figure 11: *Energy consumption of GraphViz modules*

The Influence of CPU Execution on the Energy Consumption Values During the instrumentation and data collection of the GraphViz application several tests were made. During these tests it was discovered that for some functions, the consumption values would increase in about 1000%. In a quick checkout to discover what was happening it was evident that something went wrong, and it was not a bad design of the function code. What we discovered was that when the program was executing, if the processor was working on one particular function that demanded large computational resources, the processor would be set to 100% of its capabilities. When the CPU is running at full power it consumes more energy. Therefore, the functions processed by the CPU when it was working at the maximum level had higher consumption values. The fact that this was happening had an impact on other functions besides the ones that needed such resources. The functions that lead to this suspicion were, in fact, being influenced because when the processor finished processing the resource demanding functions and started processing other functions, it was still working at high level when this was not probably needed.

The operating system is responsible for operating the hardware components of the computer. Because the OS used in these tests was developed to improve the performance of execution of a program in terms of execution time and resources, and not in terms of energy consumption, the operating system set the CPU frequency to achieve better execution times and not better energy consumption values.

In the process of trying to obtain more information about this situation, further research was made. The demanding resources functions were identified and a new instrumentation process of the software was made. In this new instrumentation, and besides the instructions to collect the energy usage, an instruction to force the program execution to pause was added. After compiling this new version and execute it again, the new data was collected. The results of this instrumentation were somewhat positive but not conclusive:

- 19% of the functions that were firstly influenced had their consumption back on the normal values.
- 15% of the functions that were firstly influenced increased their overall consumption values.
- in the other cases there was no influence.

Because of this new instrumentation, the time that the program took to execute the input obviously increased but the energy consumption values were the same as before because the energy usage was not being tracked while the program was paused.

These first results seem promising and would require more investigation. Because the optimization of the OS to energy consumption goes out of the scope of this Thesis, this topic was not explored any further, and is left as future work.

3.2. Results Treatment

The results produced by the execution of the program instrumented and compiled, are not structured which difficults the task of extracting knowledge about each component's energy consumption. Therefore, there is a need to build a structure that holds the information hierarchically and allow easy transformations to be made and immediate information calculation for each component.

The input to this phase is the output from the instrumented program execution. Thus, one needs to define a specific syntax for the input that the instrumentation output must follow.

The input syntax of this phase was defined using a grammar and is presented in Appendix A. This grammar defines the input as being a sequence of components. Each component is identified by its beginning (*Component-begin*) and its end (*Component-end*); between this it may contain more components. Inside the component begin and end is the information retrieved about its execution. Listing 2 shows a sample of this input following this grammar rules.

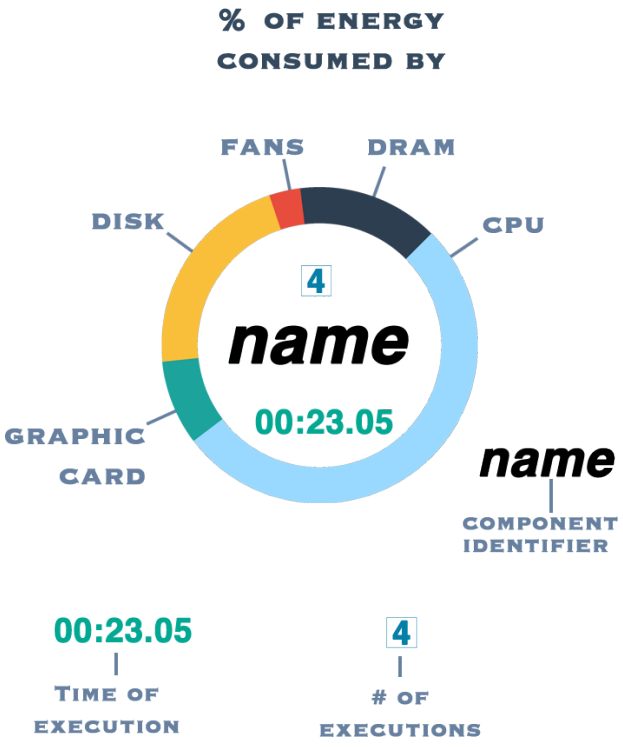


Figure 12: *Example of collected data node's information*

Having the input in a standard representation one can process this data and construct the structure needed to treat the information. This structure, and because the execution information is a hierarchy information (execution path), the representation chosen was a n-ary tree where the nodes represent the components identified, and characterized by the execution information.

In this new representation, each node contains information about the energy consumption as well as the time consumed and the number of times it was performed (a graphical representation can be consulted in Figure 12).

Listing 4: *Example of the output of the result treatment phase applied to the largest of three number program, with 6 tests and only 3 components.*

```
                  /*c1*/                                  /*c2*/                                  /*c3*/
[ [ time = 7, cpu = 65, numberUsed = 1 ]                    _                                  [ time = 9, cpu = 62, numberUsed = 1 ] ]
[ [ time = 4, cpu = 49, numberUsed = 1 ]                    _                                  [ time = 6, cpu = 63, numberUsed = 1 ] ]
[ [ time = 5, cpu = 65, numberUsed = 1 ]                    _                                  [ time = 9, cpu = 62, numberUsed = 1 ] ]
[ [ time = 7, cpu = 47, numberUsed = 1 ] [ time = 8, cpu = 31, numberUsed = 1 ]                   _                                  ]
[ [ time = 6, cpu = 65, numberUsed = 1 ]                    _                                  [ time = 9, cpu = 62, numberUsed = 1 ] ]
[ [ time = 5, cpu = 43, numberUsed = 1 ] [ time = 7, cpu = 50, numberUsed = 1 ]                   _                                  ]
```

An graphical representation of a program output example illustrating the complete tree-structure including all nodes can be seen in Figure 13. This tree-structure is built for each test run.

The next phase of the analysis of the software energy usage, described in Section 3.3, receives as input the program spectrum (containing the execution time, energy consumption and number of times used for each component in each test). So in order to produce the next phase input we must transform our n-ary trees into that program spectrum information. To do so, we start by analysing every n-ary tree. For each one, we analyze every node of the tree and collect all of its children information – this will aggregate, for each node (representation of a component), its totals (execution time, energy consumption, and number of times used). Then, having all the nodes information aggregated, we start to produce the program spectrum. Listing 4 has a sample of an output matrix of this phase. Each component test element has its execution information within the square brackets. This output (a matrix) follows the syntax grammar defined in Appendix B which is the syntax needed by the next phase, and therefore, can be used as its input. Algorithm 2 illustrates this process.

For the n tests, and for each node in the tree, a transformation to feed the next phase will be made. In this transformation each tree node aggregates all of its children information. For each test this process produces a row with all the components and its aggregated information. This output must be in a specific and standard format to be passed onto the next phase.

This phase is the following step in the methodology that we started to build in the previous Section (Section 3.1 - Figure 9). Adding it to the activity diagram we obtain the activity diagram represented in Figure 14.

To summarize we have defined the first two phases of the entire methodology:

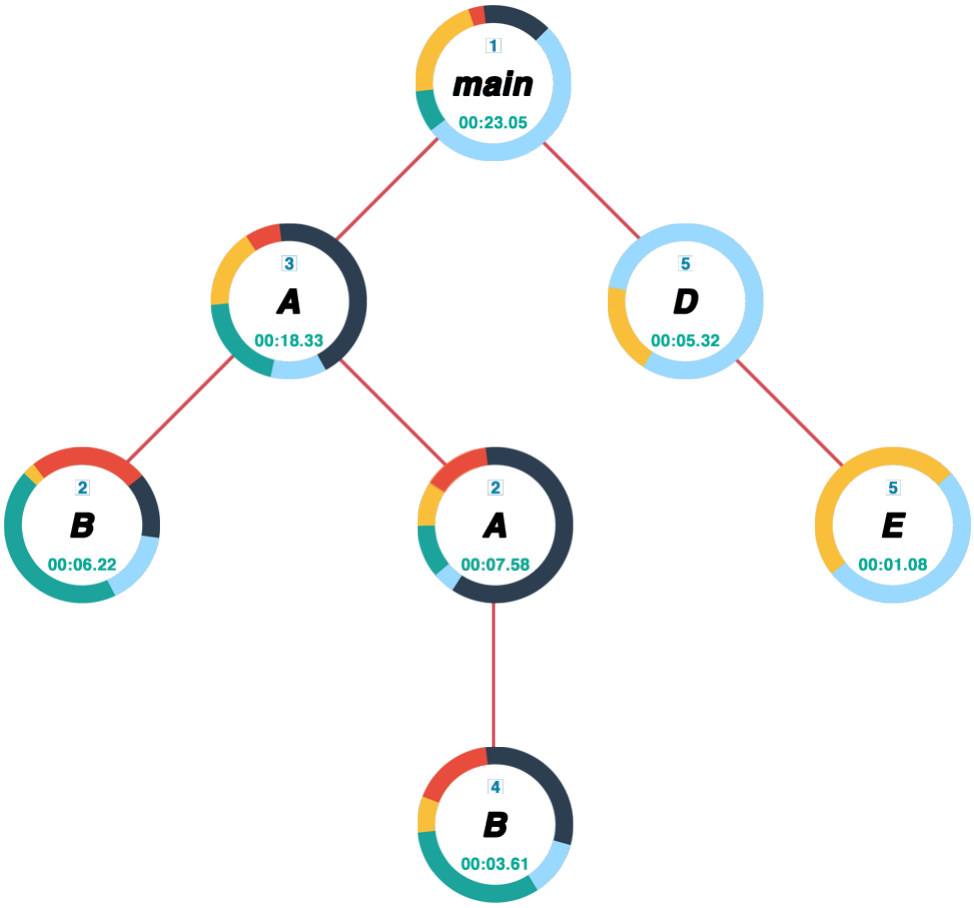


Figure 13: *An example of a n-ary tree constructed to a test's data collected. This Figure illustrates the hierarchy between calls and its consumptions*

Algorithm 2 Results treatment of the output of a instrumented program execution

```
1:  procedure TREAT THE RESULTS
2:
3:      for all inputSample in input do
4:          tree ← parseTree(inputSample)
5:          trees ← trees + tree
6:
7:          for all node in tree do
8:              component[node.name] ← component[node.name] + node.information
9:
10:         for  i ← 1 .. component.length do
11:             resultsTreatmentOutput ← resultsTreatmentOutput + component[i]
12:     return resultsTreatmentOutput.
```

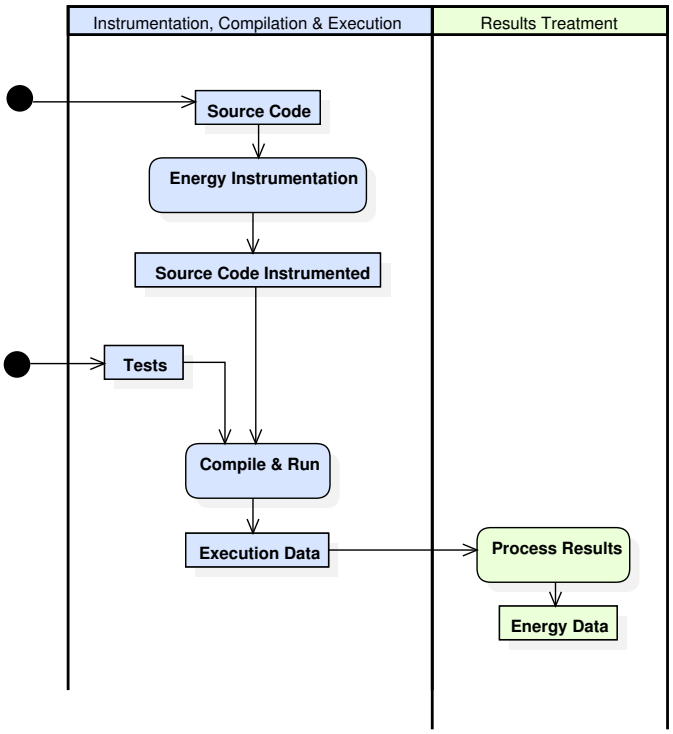


Figure 14: *Process of the methodology being constructed, containing the instrumentation, compilation and execuiton, and the results treatment phase*

- The first phase, dependent of the language/paradgim, where the program is annotated and ran with a test suite and produces the results;
- The second phase, independent of any language/paradigm, where the results from the first phase are collected and adapted in order to produce data to the analysis phase, that we are going to approach in the next Section.

3.2.1. Process Instantiation

This phase was developed and implemented in Java, and so, the grammar that this phase uses to define the input's syntax was implemented using the ANTLR framework [Parr and Quong, 1995] (when dealing with Java is one of the most used frameworks to deal with grammars). The semantic rules of this grammar were used to transform the collected data into a n-ary tree. Section 4.2 has complementary information about this implementation.

3.3. Energy Consumption Data Analysis

The technique here presented, Spectrum-based Energy Leak Localization (SPELL), is a technique that is independent of the programming language which means it is generic and therefore can be applied to different languages/paradigms.

As seen in Section 2.2.1 and as the name indicates, SFL is based on the program execution hit-spectrum. In the technique developed in this Thesis, a part of the knowledge used is also the spectrum of the program's execution. This spectrum allows the discrimination of the component usage, was it used or not, and in the cases where it was used, to extract more information about its execution. As in the SFL, the tests are also independent, i.e., the execution order of the tests is irrelevant because the state of a test does not affect the execution of another test. However, and contrary to what the SFL states, where there is an oracle to which one can ask questions about the validity of the output obtained by running a test, the SPELL analysis does not receives an oracle as input. This can be explained because, energy wise, if on one hand, there is still no known oracle to answer with 100% certainty to what is a excess of energy consumption, on the other hand, what can really be seen as an excess of energy consumption? Therefore, the oracle is not an artifact that can easily be obtained as an input.

3.3.1. The Static Model Formalization

Aside from the difference in the use of an oracle provided in the input, the technique presented here has important and complementary information to the spectrum of the execution that SFL does not need. This information can and is used as a way to obtain a more useful and complete analysis about energy consumption of the programs components.

The input of this tool is a matrix A that has n lines which correspond to the number of tests run and has m columns that are the m program's components (defined at the granularity level of the instrumentation) (Figure 15).

Each matrix element, λ_{mn}, if used in test n, contains information about the component

$$
\begin{array}{c} \\ n \text{ spectra} \end{array}
\begin{array}{c}
\begin{array}{cccc} m \text{ components} & & & \end{array} \\
\begin{bmatrix}
\lambda_{11} & \lambda_{12} & \cdots & \lambda_{1m} \\
\lambda_{21} & \lambda_{22} & \cdots & \lambda_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
\lambda_{n1} & \lambda_{n2} & \cdots & \lambda_{nm}
\end{bmatrix}
\end{array}
$$

Figure 15: *The spectrum-based energy leak localization input matrix (A)*

m execution, or nothing otherwise, as shown in Equation 2.

$$
\lambda_{ij} = \begin{cases} \left(E_{\text{energy}}, T_{\text{execution}}, N_{\#}\right)_{ij} & \text{if } c_j \text{ was used} \\ \\ \emptyset & \text{if } c_j \text{ was not used} \end{cases} \tag{2}
$$

This component's execution data is segmented in 3 categories: energy consumption, execution time and number of times executed. In the energy consumption category, values of the energy consumed by different hardware components may be present: CPU (E_{CPU}), DRAM memory (E_{DRAM}), fans (E_{fans}), hard drive (E_{disk}) and graphic card (GPU) (E_{GPU}) (Equation 3).

$$
E_{\text{energy}_{ij}} = \left(E_{\text{cpu}}, E_{\text{DRAM}}, E_{\text{fans}}, E_{\text{disk}}, E_{\text{GPU}}\right)_{ij} \tag{3}
$$

All hardware components that consume energy may have its component represented on this tuple, but on this Thesis we defined only this components because they are representative of the differences between computers. All the energy consumption values are expressed in a multiple unit of the energy unit (J): *milliJoule* (mJ). The component's execution time is represented in the attribute $T_{\text{execution}}$, this attribute is expressed in milliseconds. Finally, information about the number of executions (cardinality) is defined in $N_{\#}$ and is dimensionless.

3.3.2. The Definition of an Oracle

With the matrix that contains, for each test, the execution information of each program component, the next steps are the processing and analysis of this information.

The ideal situation would be to have an oracle adapted to the context of energy to better understand the values of the execution data and validate the correctness of a test (or as it is used in the SFL, the error vector). Thus, since we cannot get the oracle as input, the first phase will be to build one that can be used. For the oracle construction several options can be considered. The first approach could be to implement a simple metric to calculate the average energy consumption of the program in all the tests and the oracle would determine if each test consumption was above average to be recorded as a failure, and otherwise to be considered as a pass. However, this technique has some limitations, as the average energy consumption could hide significant statistical differences, one would also be ignoring the other execution information such as execution time and the number of times the component was involved in the test.

Another possibility for this oracle would be to build a base of prior execution consumption knowledge and use various programs to feed this knowledge base. The knowledge base could be segmented by type of software (image processing, graphs, etc.) and could be a correspondence between patterns of software execution and energy consumption. However, despite many positive points, the construction of this knowledge base would need a big corpus of different programs and for each one it would be necessary to catalog its execution pattern and the respective consumption. Another disavantage is that the oracle would not be independent of the input tests for which the patterns were identified and that might differ between different tests.

Thus, the solution defined for the oracle creation must be premised on the fact that it has to be relative to the program implementation and use all available information to extract the best knowledge. Another point to consider is that the oracle cannot decide with a binary criterion (fail, pass) a test execution; the criterion has to be a continuous value to represent the *greenness* of a test.

Taking the example of what is usually done in the regulation of greenhouse gas emissions

of world countries, where after assessing how much is the total emission of gases in the different years, depending on what each country contributed in gas emissions in those years, assigns the percentage of responsibility to each country. In this analysis one can try to establish an analogy, where the years are the different tests, the countries are the different components with the total for each category (energy, execution time, and cardinality), and the goal is to try to assign responsibilities to each component comparing with the total value.

To construct the oracle is then necessary, for each test, sum up all the values of the categories creating a total element. This total element will have the same structure of each component, having in each category the value resultant of the sum of all the category values (Equation 4).

$$\sum_{i=1}^{m} \lambda_{1i} = \Big(\sum_{i=1}^{m} E_{\text{energy}}, \sum_{i=1}^{m} T_{\text{execution}}, \sum_{i=1}^{m} N_{\#} \Big)_{1j} \tag{4}$$

Because this is done for the n tests at the end we will get a vector here called t (total), as shown in Equation 5.

$$\begin{bmatrix} \sum_{i=1}^{m} \lambda_{1i} \\ \sum_{i=1}^{m} \lambda_{2i} \\ \vdots \\ \sum_{i=1}^{m} \lambda_{ni} \end{bmatrix} = \overset{t}{\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix}} \tag{5}$$

Having the oracle defined, the final model that this Thesis introduces is totally defined and can be seen in Figure 16. It is based on this model that the remaining process of analysis will focus and is where from the knowledge will be extracted.

3.3.3. Analysis on the Model Using the Oracle

With the complete model established, the next step is, and following the analogy of gas emissions, relate the data of each component with the total data, as represented in Equation 6. In the end, one wants to obtain a simple structure that contains the similarity

$$\begin{array}{c} \\ n \text{ spectra} \end{array} \begin{array}{c} \overset{m \text{ components}}{\begin{bmatrix} \lambda_{11} & \lambda_{12} & \cdots & \lambda_{1m} \\ \lambda_{21} & \lambda_{22} & \cdots & \lambda_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{n1} & \lambda_{n2} & \cdots & \lambda_{nm} \end{bmatrix}} \end{array} \quad \overset{t}{\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix}}$$

Figure 16: *The spectrum-based energy leak localization input matrix (A) and the total vector (t)*

of any component (c_i) and the total vector (t).

$$\overset{c_i}{\begin{bmatrix} \lambda_{1i} \\ \lambda_{2i} \\ \vdots \\ \lambda_{ni} \end{bmatrix}} \overset{?}{\approx} \overset{t}{\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix}} \tag{6}$$

Similarity of each Category

The similarity between component i and the total vector t can be seen as how much component i is responsible for each execution information of the total vector. This association has as domain the current model and data, and therefore does not depend on prior knowledge, and is independent of other software, allowing conclusions regarding the software developed. Thus, it eliminates the dangers that could be introduced by comparing a program consumption with the consumption of other programs, since energy consumption is relative and it is totally dependent on what is the purpose of the program execution. As it would be expected, if there are few number of components, every value of each component will have bigger influence in the total vector value, which then influences the extracted similarity. The quality of the test suite is also important because only with tests that provide global coverage and test the program for different inputs, one can hope to extract interesting information.

To obtain the component similarity (ϕ) with the oracle vector, there is a need to define a function that receives the vector of a component and the total vector, and returns a structure

with the similarity (α) for each of the constituents of component's execution information (Equations 7 and 8).

$$\text{similarity}\left(\begin{bmatrix} \lambda_{1i} \\ \lambda_{2i} \\ \vdots \\ \lambda_{ni} \end{bmatrix} , \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix} \right) = \phi_i \tag{7}$$

where,

$$\phi_i = \Big(\alpha(E_{\text{Energy}}), \alpha(T_{\text{execution}}), \alpha(N_{\#}) \Big)_i \tag{8}$$

The chosen formula to calculate the similarity coefficient for each of the component's constituents, is the Jaccard similarity coefficient [Real and Vargas, 1996]. This formula, with two vectors $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$, where $x_i, y_i \geq 0$, calculates the similarity coefficient using formula present in Equation 9.

$$J(x, y) = \frac{\sum_{i=1}^{n} \min(x_i, y_i)}{\sum_{i=1}^{n} \max(x_i, y_i)} \tag{9}$$

The Jaccard similarity coefficient is a well-known formula to calculate the similarity coefficient between two vectors and has been used for a long period of time in the biology domain [Rousseau, 1998; Dombek et al., 2000].

With the application of this similarity function to all components of the matrix, the result will be a row vector that represents, for each component and for each execution, the information about their influence in the overall context. As already mentioned, this vector contains the similarity of each execution information for each component, which allows the similarity analysis to focus on a specific execution information. So, defining a sort criteria and sorting the similarity vector allows to better understand which are the components with that information that are closer to representing the totality of execution information (Equation 10. Thus, and relating to the sorting criteria, one can realize what and why are

the possible failures at energy level of the program.

$$\text{sortBy}\big(\phi, E_{\text{cpu}}, T_{\text{execution}}\big) = \big[\,\ldots\,\big] \tag{10}$$

Global Similarity

With this similarity execution information of each component it is possible to make a parameterized analysis, however, and complementary it can also be useful to have a value that encodes all the execution information. This value will allow a numerical and global comparison between the different components. This analysis will do the sorting of all components, where the components with highest value were likely to be faulty at energy level. To allow this conversion, a function that translates the execution information in a numeric value must be developed. This function aims to convert the information available into a value, which is dimensionless and therefore is not directly related to any of the units of information used. To obtain the desired value, one needs to sum all the values within the same category (Equation 13) and then multiply all the values of each category (Equation 11). The decision to multiply all categories is due to the fact that it makes the final value to grow depending on the proportion that each category adds: the higher the value of the category the higher is the proportion that it increases the overall value. Regarding the information within the same category, they have a summative contribution within the category, and will influence in proportion the global value.

$$\text{globalValue}\big(\lambda_{ni}\big) = E_{Energy_{ni}} \times T_{execution_{ni}} \times N_{\#_{ni}} \tag{11}$$

The Factor of each Energy Information

In the energy category, there are different types of results on the hardware components' energy consumption. These hardware components have a usual power consumption value and it varies from hardware component to hardware component. Therefore, it makes sense that these energy information are standardized according to the spontaneity of those hardware components to produce more power. A illustration of this normalization can be for instance:

- Two hardware components A and B, wherein A in average consumes more power than

B;

- Two software components 1 and 2 with the same total energy consumption value;
- The software component 1 energy, accounts only for the use of hardware component A;
- The software component 2 energy, accounts only for the use of hardware component B;

Besides having the same consumption value, software components 1 and 2 should have their global similarity value influenced in different ways. Because hardware component A has a higher average power consumption, software component 1 it is likely to contribute more to energy consumption in different occasions (even if for the given test suite its energy consumption value is the same as software component 2).

To apply such standardization a multiplier factor can be defined for each hardware component. Table 2 explains the average power consumption for each component[4] and the factor that it will have on the formula. This factor of an hardware component k is calculated using the formula shown in Equation 12.

$$\text{factor}_k = \frac{\text{power}_k}{\sum_{i=1}^{n} \text{power}_i} \tag{12}$$

where $power_k$ represents the average power consumption of the hardware component k, and n is the number of hardware components available.

Table 2: *Average power consumption for each hardware component*

Component name	**Power consumption (average) (W)**	**Formula factor**
CPU	102.5	0.34
DRAM	3.75	0.01
Fans	3.3	0.01
Hard Drive	7.5	0.02
GPU	187.5	0.62

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

So, with the data from Table 2 one can produce the formula present in Equation 13, to calculate the energy category of the global value.

$$E_{\text{Energy}_{ij}} = 0.34 \times E_{\text{CPU}_{ij}} + 0.01 \times E_{\text{DRAM}_{ij}} + 0.01 \times E_{\text{fans}_{ij}} + 0.02 \times E_{\text{disk}_{ij}} + 0.62 \times E_{\text{GPU}_{ij}} \quad (13)$$

With this informations the full model and its operations are specified.

This analysis hold up an important and crucial step in the methodology that is being defined. So, adding it to the respective activity diagram the definition of the methodology is concluded. In Figure 17 the complete methodology can be seen, where are identified the three distinct phases that were defined and developed along the last three Sections:

- The first phase, dependent of the language/paradigm, where the program is annotated and ran with a test suite and produces the results;
- The second phase, independent of any language/paradigm, where the results from the first phase are collected and adapted in order to produce data to the analysis phase.
- The third and last phase, independent of any language/paradigm, where the analysis of the structured execution data is performed, and energy leaks of the software are identified and can be investigated.

In the following Section an example using the analysis technique is given.

3.3.4. An Example

To understand how this analysis works and see how the analysis handles the execution data, we will present an example.

Let us think of a program that could be written in any language. This program has four different components (for instance functions in C, modules in C, methods in Java, etc.), and is ran with a test suite of five different inputs. This program has previously been through the first two phases of the methodology defined (Sections 3.1 and 3.2), and its energy, execution time, and usage have been identified. Therefore, we can use the information of this program's execution and start the analysis. In Table 3 we can see the entire model of the SPELL analysis already defined, but let us construct it step by step.

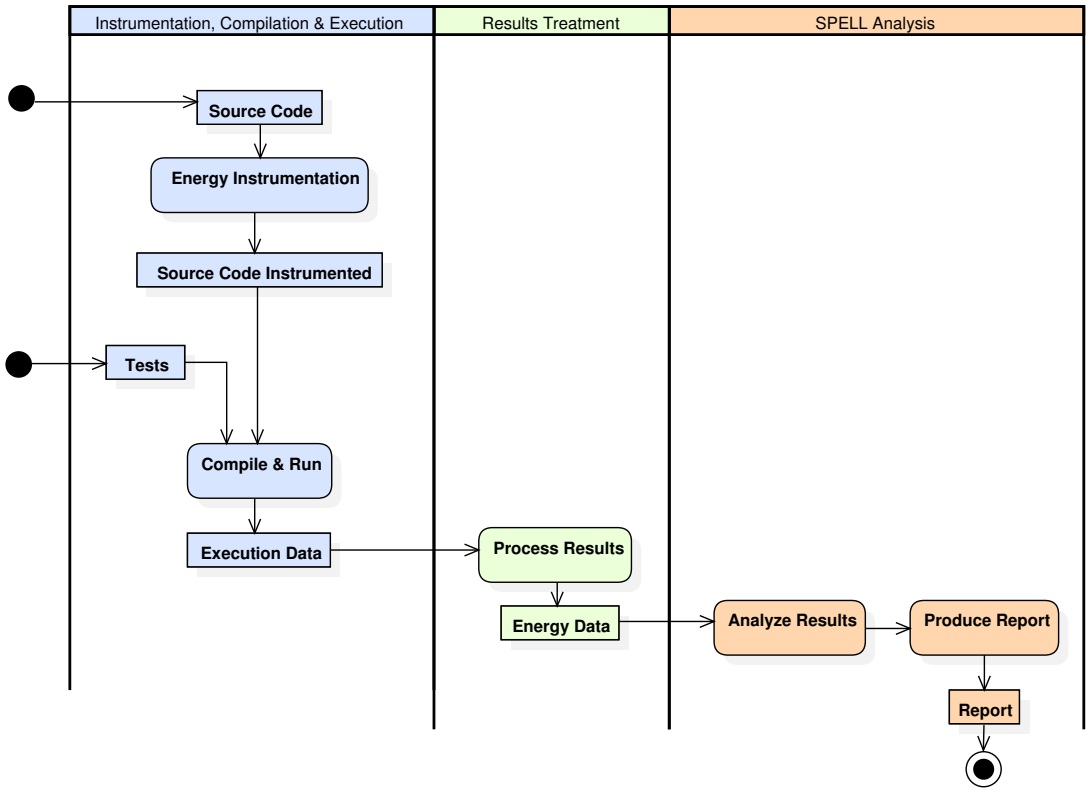


Figure 17: *Activity diagram illustrating the methodology to analyze a software to detect energy leaks*

The input data is the data seen in Table 3 where for each component and each test we have a triple of three categories. This triple contains the CPU energy consumption value (the only hardware component measured is the CPU), the number of times that software component was used, and the consumption time:

$$\begin{pmatrix} E_{CPU} \\ N_{\#} \\ T_{execution} \end{pmatrix}$$

So, in Table 3 we can check all the data from the program's execution in the given tests.

Having this inputs, and as defined in SPELL, we have to build the oracle (t vector). To do so, for each test, we sum all the values of each category of the component data. After doing this for every test we have built the "oracle vector". The following step is to calculate each component's category similarity. To achieve this we apply for each component category vector and the oracle vector the Jaccard's coefficient similarity formula. For example, for

Table 3: *SPELL matrix built for the example program*

test	c_1	c_2	c_3	c_4	t
1	$\begin{pmatrix}37\\1\\75\end{pmatrix}$	$\begin{pmatrix}61\\2\\102\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\end{pmatrix}$	$\begin{pmatrix}42\\1\\34\end{pmatrix}$	$\begin{pmatrix}140\\4\\211\end{pmatrix}$
2	$\begin{pmatrix}38\\3\\77\end{pmatrix}$	$\begin{pmatrix}50\\1\\103\end{pmatrix}$	$\begin{pmatrix}34\\2\\42\end{pmatrix}$	$\begin{pmatrix}44\\1\\37\end{pmatrix}$	$\begin{pmatrix}166\\7\\259\end{pmatrix}$
3	$\begin{pmatrix}36\\1\\73\end{pmatrix}$	$\begin{pmatrix}58\\1\\102\end{pmatrix}$	$\begin{pmatrix}35\\1\\43\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\end{pmatrix}$	$\begin{pmatrix}129\\3\\218\end{pmatrix}$
4	$\begin{pmatrix}37\\3\\74\end{pmatrix}$	$\begin{pmatrix}66\\2\\105\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\end{pmatrix}$	$\begin{pmatrix}61\\2\\43\end{pmatrix}$	$\begin{pmatrix}164\\7\\222\end{pmatrix}$
5	$\begin{pmatrix}39\\2\\75\end{pmatrix}$	$\begin{pmatrix}54\\3\\100\end{pmatrix}$	$\begin{pmatrix}51\\4\\60\end{pmatrix}$	$\begin{pmatrix}65\\2\\60\end{pmatrix}$	$\begin{pmatrix}209\\11\\295\end{pmatrix}$
similarity by component's category	$\begin{pmatrix}0.2314\\0.3125\\0.3104\end{pmatrix}$	$\begin{pmatrix}0.3577\\0.2813\\0.4249\end{pmatrix}$	$\begin{pmatrix}0.1485\\0.2188\\0.1203\end{pmatrix}$	$\begin{pmatrix}0.2623\\0.1875\\0.1444\end{pmatrix}$	
global similarity	0.0197	0.0373	0.0116	0.0112	

c_1, and for the energy category similarity coefficient we will have the formula represented in Equation 14.

$$\alpha(E_{\text{CPU}}) = \frac{min(37,140) + min(38,166) + min(36,129) + min(37,164) + min(39,209)}{max(37,140) + max(38,166) + max(36,129) + max(37,164) + max(39,209)} = 0.2314 \quad (14)$$

The calculation for the other categories and the different components would be the same, and its results can also be consulted in Table 3 in the similarity by component's category row.

To end the gathering and calculation of all the values needed to make the energy leak analysis in the program, we must calculate the global similarity of each component. To do so, we must apply the formula defined in the prior Section (Equation 11) and calculate for

each test and each component its global value.

For example, for the test 1 and the component 1, its global value would be calculated as shown in Equation 15.

$$\text{globalValue}(\lambda_{11}) = (37 \times 0.34) \times 75 \times 1 = 943.5 \tag{15}$$

Doing the same for every test of component 1 and also for each oracle test value we obtain the values represented in Table 4.

Table 4: *Component c_1 and oracle global value vector*

c_1	t
943.5	40174.4
2984.52	102325.72
893.52	28684.44
2792.76	86651.04
1989	230589.7

Using the Jaccard's coefficient similarity formula we can obtain the following similarity coefficient: 0.0197. Doing this calculations for every component of the program the global value similarity coefficient can be consulted in Table 3.

Now that we have all the needed information to analyze, we can extract some information. Reading the global similarity coefficient value we can see which component has the highest probability of have an energy leak. Sorting the components for this metric we obtain the following configuration: c_2, c_1, c_3 and finnaly c_4. This means that if the reader was a developer of this application he/she should consider looking first into the component c_2 to improve the energy consumption of the program. The advantage of the SPELL technique is that it can tell, besides the global value, why the component is faulty. For example, c_2 is calculated as the most probable component to have a energy leak because if we look into its categories similarity values we will see that this component ranks first in the energy similarity value, second in the cardinality similarity value and first in the execution time similarity. This ranks clearly points to this component. Also, there are some curious facts that can be seen in this analysis. For example, c_4 has an energy category similarity value higher than the c_1, although and due to the other categories its ranked fourth in the overall.

Other curiousity is that c_3 has in test 5 an higher value of energy consumed than any of the c_1 energy values retrieved. However and because we take into consideration multiple tests, c_3 is ranked third in the overall, when c_1 is ranked second. Other curious facts could be found and explained but, and to compare this analysis over a technique using only the energy consumption values, another fact will be given. If we calculate the components average energy consumption values we would obtain:

$$c_1 = 37.4,\ c_2 = 57.8,\ c_3 = 24,\ c_4 = 42.4$$

what would indicate the following ranking: c_2, c_4, c_1 and finnaly c_3. This rank is completely different from the obtained in the SPELL analysis because it ignores the other components influence. To prove that this technique produces true conclusions in the Chapter 5 a validation of the technique will be presented.

3.3.5. Process Instantiation

This last phase was also developed and implemented using Java. Therefore, the grammar for the input was created using the ANTLR framework. Every operation described in this phase (e.g. the calculation of each category similarity), were implemented as Java methods. More details about this implementation can be consulted in Section 4.3.

4. The SPELL Framework

Throughout the Thesis development, all the phases already identified in Section 3.1, 3.2 and 3.3 were materialized as individual tools. Doing so allowed us to use them separately and if needed to be modified and updated without having to propagate those changes to the other phases. Another factor that weighed in the decision to build multiple tools was the compilation process. For a program to be able to use the Intel Power Gadget framework to retrieve the energy consumption information, a compilation flag needs to be added to the compilation process. Because in C developed software, every program has its own *makefile* (file that builds the software from its source files), this process, being a specific process for each program's makefile, could not be systematized.

To combine the multiple tools that we created we used the CROSS platform introduced in [Martins et al., 2012] - a web portal that allows the construction of certifications[5] to analyze open source software using different tools. In this portal, one can create a certification to represent the whole process of this tool which will be automatically linked. A possible certification for the whole process using the modules built in this Thesis is represented in Figure 18. In this Figure 18, there are six tools, represented by six boxes. Each tool has an input language that for each input given, transforms it into the output language – this input and output languages are also represented in these boxes (*inputLanguage* → *outputLanguage*). The connection between the boxes represents the flow that the input will follow, and the transformation that will suffer, when submitted to perform the certification. In this certification, the first tool is the energy instrumentation, that will receive and produce a program in the C language; then it is compiled by the gcc; executed by *softwareExecutor*, that produces the output of the execution; its results are then treated by the next tool (*text2SPELLInput*), which produces a matrix to be used in the energy analysis phase; in the next phase the energy analysis is performed on this data by *SPELLAnalysis*; and, in the end, a tool transforms the output report from the analysis into a formatted report by *text2Report* (this report is a CROSS language specification and must be the last type of the certification process). This visual language was defined by us and is presented in [Carção

[5]Certifications are programs that run software throught a set of tools and analyze it.

and Martins, 2014].

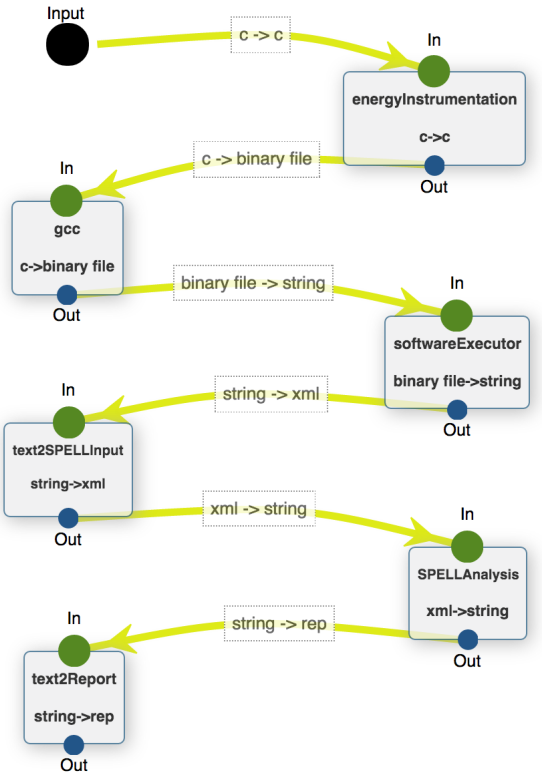


Figure 18: *A visual certification that represents the composition of the different modules to create a full process.*

This set of tools is enclosed in a framework. The deployment diagram, the UML diagram most suitable to represent a framework's modules architecture, shown in Figure 19, illustrates this framework composition and its tools (known in the deployment diagram as components). Each component identifies an executable tool and contains its implementation language.

In the following Sections each tool of the framework will be explained in detail.

4.1. The Instrumentation, Compilation and Execution

As it was described in Section 3.1.3, this phase was implemented to instrumentate, compile and execute C programs in order to analyze C programs.

This was developed in C++ using the Clang framework to retrieve and annotate the AST (Figure 19 - *ClangAST Instrumentation*). The Power Gadget Tool from Intel was the

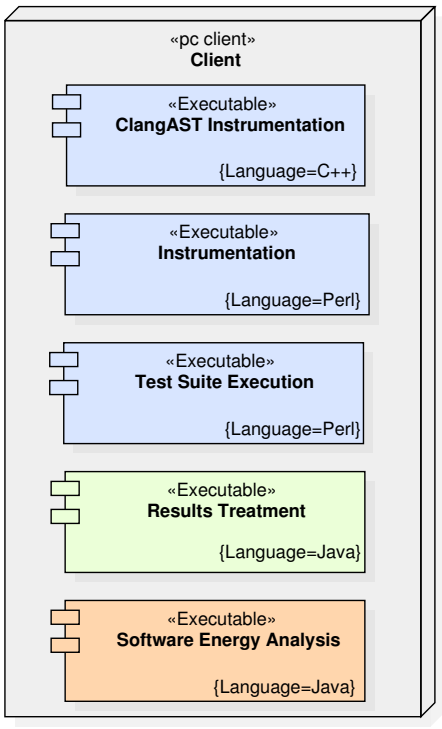


Figure 19: *Deployment diagram of the SPELL framework developed*

source of the energy information. In the end, a Perl script that applied the instrumentation to every module of a software package was developed (Figure 19 - *Instrumentation*). To execute this package (now instrumented) was also developed a Perl script (Figure 19 - *Test Suite Execution*).

Because the RAPL tool has a limitation that do not isolates the processes, which means that in its energy measures it takes into account the whole system, we took some cautions. To try to exclude the system influence over the program energy consumption values, the test was ran multiple times (40 in our case), where the 10 runs with the higher consumption were discarded. With the rest of the executions, an average for each consumption value was calculated.

The data produced by this tool can be used in the tool of the next Section.

4.2. The Results Treatment

As stated in Section 3.2.1, this tool (Figure 19 - *Results Treatment*) was developed in Java, and its grammar was implemented using ANTLRWorks framework. The software architecture of this tool is shown in in Figure 20 and is represented in a class diagram. This class diagram contains the different packages of the tool and its classes.

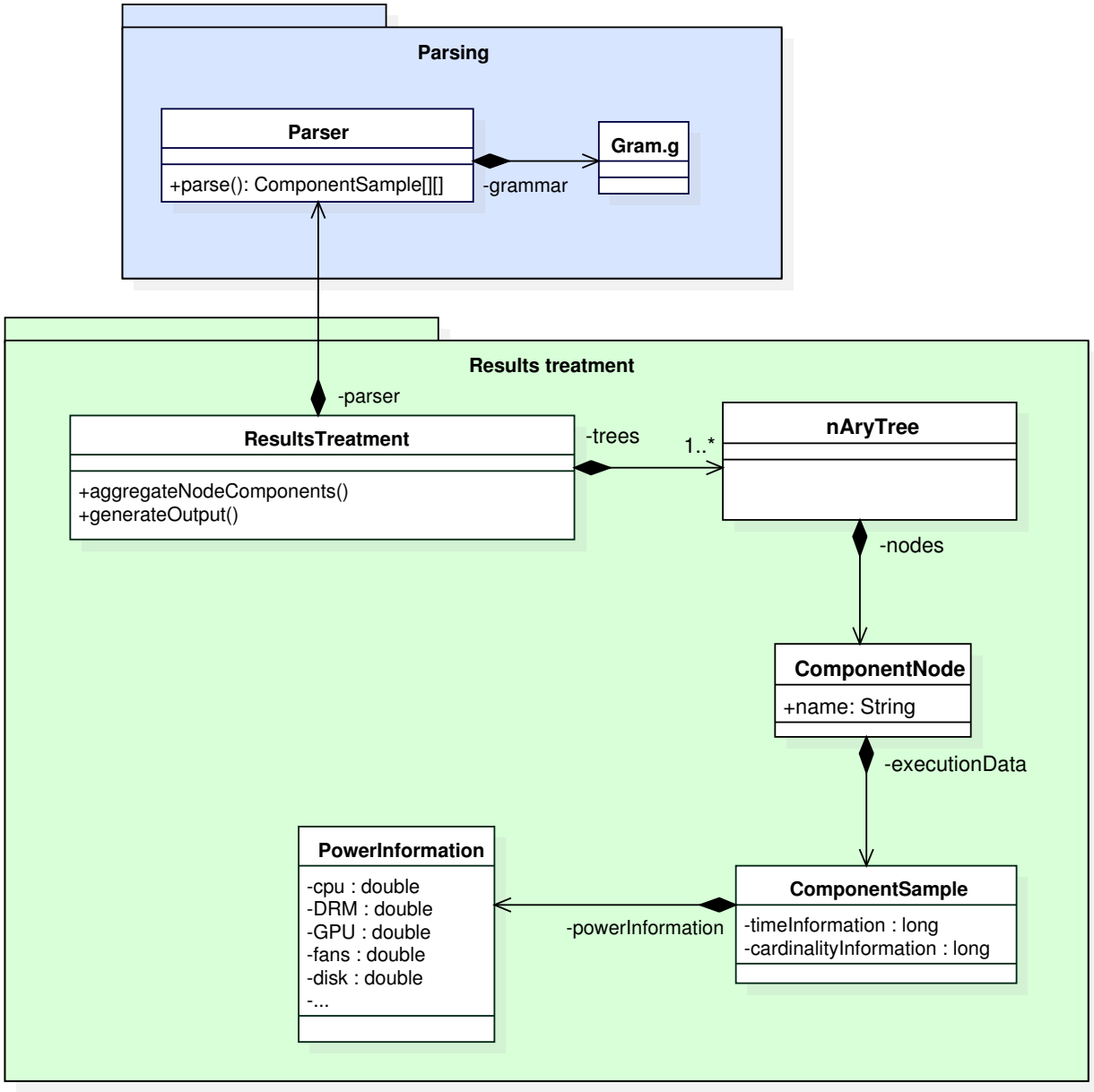


Figure 20: *Class diagram illustrating the internal design of the results treatment tool*

With the n-ary tree that represents the execution path of each of the program's component build, various operations are performed on the tree. Each component can appear multiple times as a node on the tree, so, its information is aggregated and refreshed. Traversing the tree and having done this for all the components, the information aggregated is ready to be produced to the next phase. In a similar way the next phase's input must also follow a specific syntax.

4.3. SPELL Analysis

The last tool (Figure 19 - *Software Energy Analysis*), and as shown in Section 3.3.5, was also developed in Java using the ANTLRWorks framework to implement its grammar.

As this tool implements the SPELL analysis it must implement its concepts. Each concept and its corresponding Java artifact is represented in the Table 5.

Table 5: *Correlation between the SPELL concepts and its implementation in the tool*

SPELL concept	**Implemented as**
Software Component's power information	**Class:** $PowerInformation$
Software Component	**Class:** $ComponentSample$
Matrix of software components	**Instance Variable:** $ComponentSample[\,][\,]$
Oracle	**Instance Variable:** $ComponentSample[\,]$
Formula to calculate the similarity coefficient	**Class:** $SimilarityFormula$
Formula to apply the calculation of the similarity between the component and the oracle	**Class:** $ComponentSimilarityStrategy$
Component similarity	**Class:** $ComponentSimilarity$
An array of components' similarity	**Instance Variable:** $ComponentSimilarity[\,]$
Global value of component's	**Class:** $TotalValueComponent$
An array of components' global value	**Instance Variable:** $TotalValueComponent[\,]$

In this tool, there is a main class (SPELLAnalysis) that is the center of this process. It has the model information and the operations of this model (defined in Section 3.3, i.e., calculating the oracle, sortBy criteria, and compute global value).

The software architecture of this module is represented in the class diagram presented in the Figure 21.

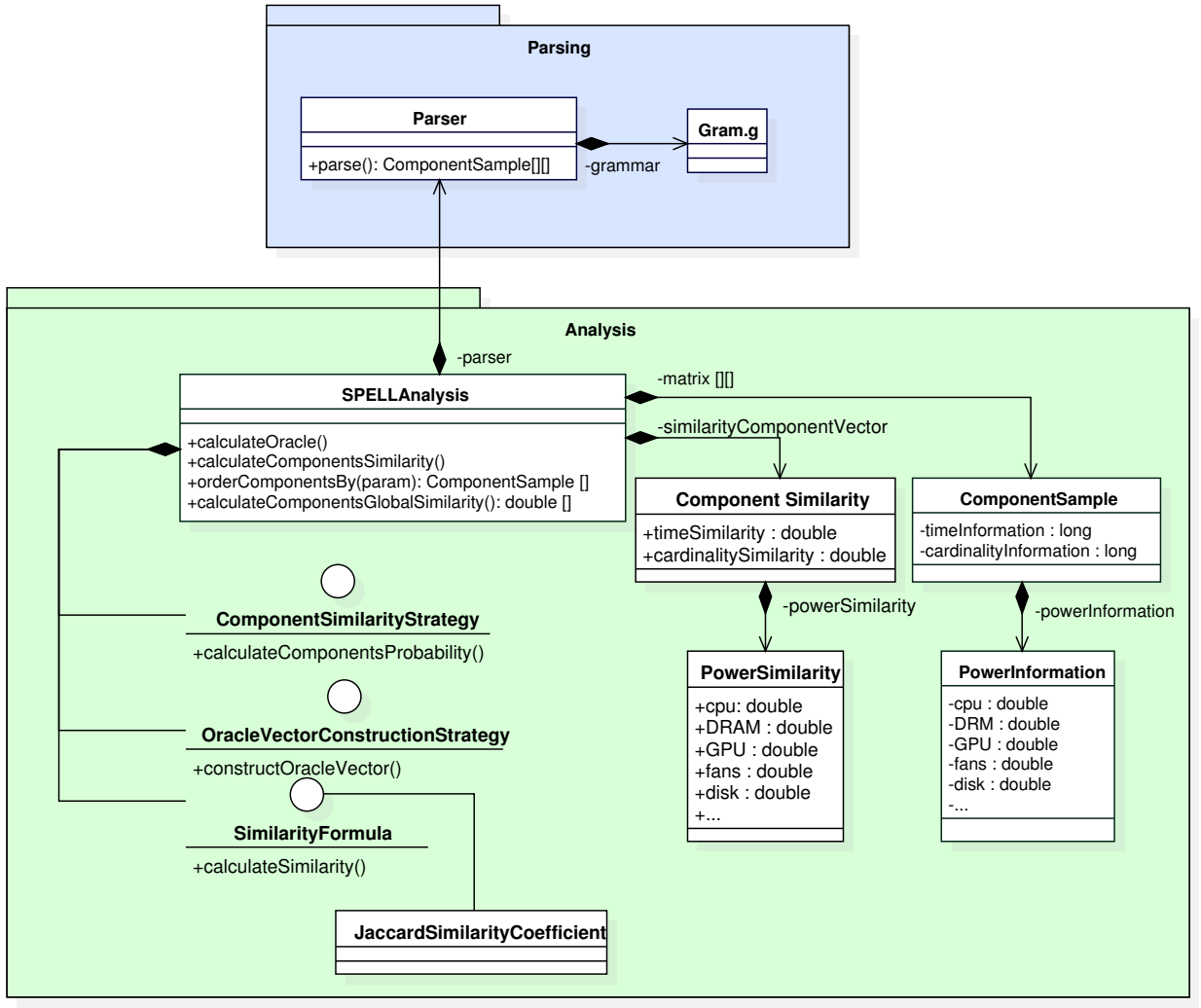


Figure 21: *Class diagram illustrating the internal design of the Software Energy Analysis tool*

4.4. How to Use the SPELL Framework

This framework can be obtained at `https://github.com/tcarcao/spellframework`. The package contains all of three tools and a README file. These tools must be ran separately or linked all together in a platform like the CROSS portal. The README file gives instructions on how to run each of the tools and the pre-requisites to run them.

5. Validation

To evaluate the results reported by the methodology and the framework developed, we need to, and using our framework, find energy leaks that are already known.

In Section 2.1 we already saw that Gutiérrez et al. [2014] made a research on how much the Java collections consumed, and made an energy consumption rank of those collections, identifying the collections that perform best on energy usage. To build this rank they ran an well-known benchmark[6] and measured the energy consumption of the benchmark with the different collections. Table 6 shows the operations performed in the benchmark for each collection.

Table 6: *Operations performed in the benchmark for each collection*

Operations performed in the benchmark
add 100000 distinct elements
addAll 1000 times 1000 elements
clear
contains 1000 times
containsAll 5000 times
iterator 100000
remove 10000 elements
removeAll 10 times 1000 elements
retainAll 10 times
toArray 5000 times

The format of their conclusions were, for each collection, how many times, when replaced by other collection, the program energy consumption values got better or how many times got worse. We can transform this format in a rank (worst to better) that is represented in Table 7.

To apply our framework to the same problem we have to adapt it. Because our framework only works with C programs we have to run the Java code from a proxy C program. This program instantiates a version of the Java Virtual Machine (JVM) and then uses Java Native Interface (JNI) to run the desired code. To analyse this benchmark in terms of energy in our framework, we need to define which are the components and the execution tests. As

[6] `http://java.dzone.com/articles/java-collection-performance`

Table 7: *Rank obtained by [Gutiérrez et al., 2014], from worst to better, of the Java collections*

ConcurrentLinkedDeque
LinkedBlockingDeque
LinkedList
LinkedTransferQueue
ConcurrentLinkedQueue
ArrayList
PriorityQueue
CopyOnWriteArrayList
ConcurrentSkipListSet
TreeSet
CopyOnWriteArraySet
LinkedHashSet
HashSet

we want to know which collection is more likely to have an energy leak, the collections are defined as the components. Each operation available in the benchmark (Table 6) will be a test to the components (the collections). For each operation, the energy is only measured after initializing the JVM, thus eliminating the energy usage of the JVM initialization[7].

Applying our framework to the benchmark, the analysis input and calculated similarities (as shown in Section 3.3) can be consulted in Table 9[8]. Because each operation was called only once in the execution to simulate a test, the usage cardinality of each component element is always 1.

Table 8 contains the comparison between the rank obtained by [Gutiérrez et al., 2014] and our collections rank ordered by global similarity value. Comparing these two ranks we can observe that 9 of 13 collections have the same rank and only two pairs of collections are misplaced. It is important to mention that *ConcurrentLinkedDeque* and *LinkedBlocking-Deque*, and also, *LinkedHashSet* and *HashSet* were reported to have close values [Gutiérrez et al., 2014]. Also, in our analysis, the similarity value of energy consumption category are very alike (0.0960 vs 0.0919 and 0.0640 vs 0.0660) as well as the global similarity (0.1160 vs 0.1080 and 0.0482 vs 0.0447). Therefore, a possible lack of precision on the energy measure

[7] Using the JVM and JNI, there is an excess of consumption in each benchmark operation but because it is constant for every operation, in terms of operations comparison, this excess of consumption is negligible.

[8] Due to size constrains, Table 9 is the inverse matrix where the components are the rows and the tests (methods) are the columns.

Table 8: *Rank obtained by [Gutiérrez et al., 2014] on the left vs our analysis rank on the right*

ConcurrentLinkedDeque	*LinkedBlockingDeque*
LinkedBlockingDeque	*ConcurrentLinkedDeque*
LinkedList	*LinkedList*
LinkedTransferQueue	*LinkedTransferQueue*
ConcurrentLinkedQueue	*ConcurrentLinkedQueue*
ArrayList	*ArrayList*
PriorityQueue	*PriorityQueue*
CopyOnWriteArrayList	*CopyOnWriteArrayList*
ConcurrentSkipListSet	*ConcurrentSkipListSet*
TreeSet	*TreeSet*
CopyOnWriteArraySet	*CopyOnWriteArraySet*
LinkedHashSet	*HashSet*
HashSet	*LinkedHashSet*

may be the explanation to this difference. It is important to highlight that these differences only affect a misplace in one position and in any case this technique ranks a supposedly free energy leak collection as a collection with a high probability of being energy faulty. At most, this misplace, would lead the developer to choose a different collection, that would not impact by much its results, with the addition that he/she in our framework could know what were the reasons behind such fact.

With our analysis we came to very much the same conclusion of [Gutiérrez et al., 2014] about which Java collections were the better and the worse in terms of energy. This means that our solution works, and therefore, may be used to identify energy leaks in the software with the addition of being able to give extra reports on why is that happening.

Table 9: *SPELL Matrix built for the benchmark test. Collections are the components (rows) and the operations to the collections are the tests (columns).*

	add	**addAll**	**clear**	**contains**	**containsAll**	**iterator**	**remove**	**removeAll**	**retainAll**	**toArray**	**similarity by component's category**	**Global similarity**
LinkedBlockingDeque	796	614	918	1293	1241	1101	1387	1137	1247	1306	0.0960	0.1160
	1	1	1	1	1	1	1	1	1	1	0.0769	
	770	636	4936	5120	5212	3782	5290	4144	5183	4086	0.0949	
ConcurrentLinkedDeque	709	550	1046	1394	945	1035	1257	1399	1086	1145	0.0919	0.1080
	1	1	1	1	1	1	1	1	1	1	0.0769	
	705	619	5007	4313	5300	3728	4985	4335	5123	3983	0.0923	
LinkedList	770	524	1048	926	996	811	1008	1075	1194	1400	0.0848	0.0935
	1	1	1	1	1	1	1	1	1	1	0.0769	
	694	817	4980	2961	4607	3198	4951	4373	4764	4370	0.0870	
LinkedTransferQueue	760	722	1250	702	1226	920	1159	647	1226	989	0.0835	0.0881
	1	1	1	1	1	1	1	1	1	1	0.07692	
	787	815	4409	3219	4618	3142	5414	3513	4438	3987	0.0832	
ConcurrentLinkedQueue	741	700	1163	999	954	701	1164	810	1277	889	0.0817	0.0831
	1	1	1	1	1	1	1	1	1	1	0.0769	
	724	852	4391	3359	3838	3339	5352	3381	4396	3622	0.0806	
ArrayList	503	747	970	1024	1335	533	1106	728	961	746	0.0752	0.0776
	1	1	1	1	1	1	1	1	1	1	0.0769	
	430	2712	2785	3268	5431	1835	4442	4653	4906	2221	0.0792	
PriorityQueue	721	998	1134	908	514	1069	631	545	883	1182	0.0746	0.0761
	1	1	1	1	1	1	1	1	1	1	0.0769	
	3182	3906	4690	4503	481	3781	2449	1816	2596	4980	0.0785	
CopyOnWriteArrayList	827	1063	772	533	704	727	1089	1217	739	768	0.0734	0.0696
	1	1	1	1	1	1	1	1	1	1	0.0769	
	3678	4163	763	3121	3493	1965	4779	3866	3595	2363	0.0770	
ConcurrentSkipListSet	625	865	1016	1043	687	1362	597	566	554	964	0.0720	0.0657
	1	1	1	1	1	1	1	1	1	1	0.0769	
	2977	3566	3603	3890	618	3881	2505	2208	2060	4319	0.0718	
TreeSet	787	909	591	935	199	1020	1070	741	705	981	0.0690	0.0650
	1	1	1	1	1	1	1	1	1	1	0.0769	
	2980	3790	382	4882	370	3887	3492	1995	2244	4826	0.0699	
CopyOnWriteArraySet	1307	975	685	550	708	725	1067	742	543	520	0.0680	0.0644
	1	1	1	1	1	1	1	1	1	1	0.0769	
	5160	5070	3055	2271	435	2229	5443	2054	2048	956	0.0696	
HashSet	738	622	1042	730	896	581	706	785	636	859	0.0660	0.0482
	1	1	1	1	1	1	1	1	1	1	0.0769	
	2245	2794	3201	2001	2787	2314	2367	2214	2304	2874	0.0608	
LinkedHashSet	556	625	970	586	765	673	738	732	639	1073	0.0640	0.0447
	1	1	1	1	1	1	1	1	1	1	0.0769	
	980	2397	3240	1160	1017	3184	1968	2729	2626	3771	0.0559	
Oracle	9840	9914	12605	11623	11170	11258	12979	11124	11690	12822		
	13	13	13	13	13	13	13	13	13	13		
	25312	32137	45442	44068	38207	40265	53437	41281	46283	46358		

6. Conclusion

The importance of the energy consumption is felt either on users of mobile devices or by software developers. The programming languages for several years have been providing tools to allow software developers to improve the execution performance of their programs and eliminate faults (debuggers, profilers, etc.). In this Thesis we developed techniques and a framework that help the developers to localize energy leaks. By doing so, we hope to impact the development of green software by making it easier and productive.

To achieve this Thesis results, we first started by identifying the different steps to reach our goal. The instrumentation, compilation and execution is the first step and is where the software is transformed to produce information about its execution, and then is compiled and executed. This phase was implemented using Clang and Perl and analyzes C language programs. This is the only phase programming language dependent. In a second step, the analysis of the information produced in the first step is made and the matrix needed in step three is generated using the first step's data. This second phase was developed in Java and is language/paradigm independent. In the third and last step an analysis is performed and the conclusions about energy leaks are drawn. This last phase was developed in Java and is also language/paradigm independent. We have also developed a framework enclosing these three tools. A validation of the methodology and framework created was performed and we have shown that it accomplishes very good results.

Throughout the Thesis research and development multiple contributions were made. The main contributions of this Thesis are:

- A methodology to analyze a program's source code energy consumption.
  This methodology defines what are the steps to be taken in order to execute, read and analyze a program's energy usage.

- A software tool that allows the instrumentation to retrieve energy and execution information.
  This module allows the instrumentation of C programs to extract the execution data of the source code.

- A software tool that processes the execution data.

  This module is independent of the program language and accepts as input the program's execution data. It processes this data and aggregates the information by component.

- A software tool that analyzes the execution data.

  This module (SPELL) is also programming language independent. It analyzes the program execution data and produces as output which are the energy leaks in the program.

- A framework that encloses all the modules above mentioned.

6.1. Research Questions Answered

We proposed ourselves to answer three research questions. These questions are related to the concept, the design, and implementation of this Thesis. Now, we can answer these questions.

Q1 *Can we define a methodology to analyze the energy consumption of software source code?*

Yes, we identified three different phases (Sections 3.1, 3.2 and 3.3) in the path to analyze the energy consumption of software source code and detailed each one of them. Within this definitions we systematized the process and extracted such methodology.

Q2 *Is it possible to adapt a general purpose fault localization algorithm to the context of energy leak localization?*

Yes, in Section 2.2.1 we identified a fault localization technique which in Section 3.3 we analyzed and transformed, making the necessary changes while introducing other useful concepts, having reached an energy leak localization technique.

Q3 *Can we find energy leaks in software source code?*

Indeed, after defined and implemented the energy leak localization technique in Section 3.3, in Chapter 5 we automatically identified energy leaks that in fact were energy leaks known.

6.2. Other Contributions

In addition to this Thesis contributions, during its work I was involved in other research topics that culminated in three publications and one prize award:

- **A Visual DSL for the Certification of Open Source Software**, Tiago Carção, Pedro Martins. In the proceedings of the *International Conference on Computational Science and Its Applications (ICCSA'14)*, Guimarães, Portugal, June 30 - July 3, 2014.
- **Detecting Anomalous Energy Consumption in Android Applications**, Marco Couto, Tiago Carção, Jácome Cunha, João Paulo Fernandes, João Saraiva. In the proceedings of the *Brazilian Symposium on Programming Languages (SBLP'14)*, Maceio, Brazil, October 2-3, 2014.
- **Measuring and visualizing energy consumption within software code**, Tiago Carção. In the proceedings of the *Visual Languages and Human-Centric Computing (VL/HCC'14)*, Melbourne, Victoria, Australia, July 28 - August 1, 2014.
- **Energy consumption detection in LabView**, Tiago Carção, Jácome Cunha, João Paulo Fernandes, Rui Pereira, João Saraiva. Grand prize ($2000) of a competition on innovating ideas applied to a specific software, awarded by National Instruments

6.3. Future Work

With the contributions of this Thesis as basis, where we already defined a methodology and a technique analysis that can identify energy leaks in the source code, some research could be made to further improve the tools available to help the development of software applications. This research can target the following topics:

- As noted in Section 3.3 where we choose to use the energy consumption and not the power consumption in the technique, a further investigation could be done in using the power consumption to extract results and trying to identify patterns and possibly bad software components.

- More languages could be investigated, and therefore, the phase of instrumentation, compilation, and execution, could be instantiated to those language.

- Since one of the basis of the technique developed is the function that gives the similarity between components and the oracle vector, to improve the accuracy of the technique developed, other functions of similarity could be tested.

- Identify patterns of energy usage (red smells): Having a tool to identify the energy leaks in a software program, one can run multiple software packages and identify some bad smells in terms of energy.

- Propose refactorings to remove those red smells. With the red smells identified, multiple techniques to refactor them with a greener version can be researched.

- Develop a visual tool to present the information collected. With all of the information – the energy leaks, the red smells, and consequent refactorings – we need to present this information. Thus, a visual tool, that can also be integrated in an IDE, that implement these techniques can be developed. This tool would be a major contribution to the daily tasks of the software developer.

- Research the influence of CPU execution on the energy consumption values. As described in Subsection 3.1.4, during the development of this Thesis we faced an odd situation in the energy measurement of a C program execution. We did some research and got some results that seem promising and require more investigation. A more profound investigation on trying to find a win-win situation in the execution time and the energy consumption levels, by optimizing the OS to prioritize the performance in terms of energy consumption, should be made.

References

Abreu, R. (2009). *Spectrum-based Fault Localization in Embedded Software*. PhD in computer science, Delft University of Technology.

Abreu, R., Zoeteweij, P., Golsteijn, R., and van Gemund, A. J. C. (2009). A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792.

Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2006). An evaluation of similarity coefficients for software fault localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC '06, 18-20 December, 2006, University of California, Riverside, USA*, pages 39–46.

Anthony, S. (2013). Intel: Haswell will draw 50% less power than Ivy Bridge. `http://www.extremetech.com/computing/156739-intel-haswell-will-draw-50-less-power-than-ivy-bridge`. Accessed: 2014-10-23.

Arnoldus, J., Gresnigt, J., Grosskop, K., and Visser, J. (2013). Energy-efficiency indicators for e-services. In *2nd International Workshop on Green and Sustainable Software, GREENS '13, San Francisco, CA, USA, May 20, 2013*, pages 24–29.

Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer Networks*, 54(15):2787–2805.

Brownlee, J. (2013). OS X Mavericks Will Improve Your Battery Life By As Much As 4 Hours. `http://www.cultofmac.com/251135/os-x-mavericks-will-improve-your-battery-life-by-as-much-as-4-hours/`. Accessed: 2014-09-23.

Cai, X. and Lyu, M. R. (2005). The effect of code coverage on fault detection under different testing profiles. In *Proceedings of the ICSE '05 Workshop on Advances in Model-Based Software Testing, A-MOST 2005, St. Louis, Missouri, USA*.

Carção, T. and Martins, P. (2014). A visual DSL for the certification of open source software. In *Computational Science and Its Applications, ICCSA '14, 14th International Conference, Guimarães, Portugal, June 30 - July 3, 2014, Proceedings, Part V*, pages 602–617.

Couto, M. (2014). Monitoring Energy Consumption in Android Applications. Master's thesis, University of Minho.

Couto, M., Carção, T., Cunha, J., Fernandes, J. P., and Saraiva, J. (2014). Detecting anomalous energy consumption in android applications. In *Programming Languages - 18th Brazilian Symposium, SBLP '14, Maceio, Brazil, October 2-3, 2014. Proceedings*, pages 77–91.

Crisostomo, C. (2012). Intel's Ivy Bridge Processors: The Most Energy Efficient CPU's Yet. `http://www.theenvironmentalblog.org/2012/08/intels-ivy-bridge-processors-energy-efficient-cpus`. Accessed: 2014-10-23.

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

Fanara, A., Haines, E., and Howard, A. (2009). The state of energy and performance benchmarking for enterprise servers. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC '09, Lyon, France, August 24-28, 2009, Revised Selected Papers*, pages 52–66.

Ferreira, M. A., Hoekstra, E., Merkus, B., Visser, B., and Visser, J. (2013). SEFLab: A lab for measuring software energy footprints. In *2nd International Workshop on Green and Sustainable Software, GREENS '13, San Francisco, CA, USA, May 20, 2013*, pages 30–37.

Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software - Practice and Experience (SPE)*, 30(11):1203–1233.

Gonçalves, R., Saraiva, J., and Belo, O. (2014). Defining energy consumption plans for data querying process. In *Proceedings of 4th IEEE International Conference on Sustainable Computing and Communications*, SustainCom '14, Sydney, Austrália.

Google (2014). Better data centers through machine learning. `http://googleblog.blogspot.pt/2014/05/better-data-centers-through-machine.html`. Accessed: 2014-09-23.

Grosskop, K. (2013). PUE for end users - are you interested in more than bread toasting? *Softwaretechnik-Trends*, 33(2).

Grosskop, K. and Visser, J. (2013). Energy efficiency optimization of application software. *Advances in Computers*, 88:199–241.

Guelzim, T. and Obaidat, M. S. (2013). Chapter 8 - Green Computing and Communication Architecture. In Obaidat, M. S., Anpalagan, A., and Woungang, I., editors, *Handbook of Green Information and Communication Systems*, pages 209–227. Academic Press.

Gutiérrez, I. L. M., Pollock, L. L., and Clause, J. (2014). SEEDS: a software engineer's energy-optimization decision support framework. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 503–514.

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

Harmon, R. R. and Auseklis, N. (2009). Sustainable IT services: Assessing the impact of green computing practices. pages 1707–1717. IEEE.

Harrold, M. J., Rothermel, G., Sayre, K., Wu, R., and Yi, L. (2000). An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification & Reliability (STVR)*, 10(3):171–194.

Hönig, T., Eibel, C., Schröder-Preikschat, W., Cassens, B., and Kapitza, R. (2013). Proactive energy-aware system software design with SEEP. *Softwaretechnik-Trends*, 33(2).

Hurni, P., Nyffenegger, B., Braun, T., and Hergenroeder, A. (2011). On the accuracy of software-based energy estimation techniques. In *Wireless Sensor Networks - 8th European Conference, EWSN '11, Bonn, Germany, February 23-25, 2011. Proceedings*, pages 49–64.

Jain, A. K. and Dubes, R. C. (1988). *Algorithms for Clustering Data*. Prentice-Hall.

Korel, B. and Laski, J. W. (1988). Dynamic program slicing. *Information Processing Letters*, 29(3):155–163.

Li, D., Hao, S., Halfond, W. G. J., and Govindan, R. (2013). Calculating source line level energy information for android applications. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 78–89.

Li, D., Jin, Y., Sahin, C., Clause, J., and Halfond, W. G. J. (2014). Integrated energy-directed test suite optimization. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 339–350.

Martins, P., Fernandes, J. P., and Saraiva, J. (2012). A web portal for the certification of open source software. In *Information Technology and Open Source: Applications for Education, Innovation, and Sustainability - SEFM '12 Satellite Events, InSuEdu, MoKMaDS, and OpenCert, Thessaloniki, Greece, October 1-2, 2012, Revised Selected Papers*, pages 244–260.

Mayer, W. and Stumptner, M. (2008). Evaluating models for model-based debugging. In *23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, 15-19 September 2008, L'Aquila, Italy*, pages 128–137.

Mouftah, H. T. and Kantarci, B. (2013). Chapter 11 - Energy-Efficient Cloud Computing: A Green Migration of Traditional IT. In Obaidat, M. S., Anpalagan, A., and Woungang, I., editors, *Handbook of Green Information and Communication Systems*, pages 295–330. Academic Press.

Noureddine, A., Rouvoy, R., and Seinturier, L. (2014). Unit testing of energy consumption of software libraries. In *Symposium on Applied Computing, SAC '14, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1200–1205.

Parr, T. J. and Quong, R. W. (1995). ANTLR: A predicated- *LL(k)* parser generator. *Software - Practice and Experience (SPE)*, 25(7):789–810.

Pinto, G., Castor, F., and Liu, Y. D. (2014). Mining questions about software energy consumption. In *11th Working Conference on Mining Software Repositories, MSR '14, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 22–31.

Ralph, N. (2011). Lab Tested: Intel's Sandy Bridge CPUs Deliver Blazing Speed and Energy Savings. `http://www.techhive.com/article/215318/Intel_Sandy_Bridge.html`. Accessed: 2014-10-23.

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

Reps, T. W., Ball, T., Das, M., and Larus, J. R. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem. In *Software Engineering - ESEC/FSE '97, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, Proceedings*, pages 432–449.

Ricciardi, S., Palmieri, F., Torres-Viñals, J., Martino, B. D., Santos-Boada, G., and Solé-Pareta, J. (2013). Chapter 10 - Green Data center Infrastructures in the Cloud Computing Era. In Obaidat, M. S., Anpalagan, A., and Woungang, I., editors, *Handbook of Green Information and Communication Systems*, pages 267–293. Academic Press.

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

Rousseau, R. (1998). Jaccard similarity leads to the marczewski-steinhaus topology for information retrieval. *Information Processing & Management*, 34(1):87–94.

Rühl, C., Appleby, P., Fennema, J., Naumov, A., and Schaffer, M. (2012). Economic development and the demand for energy: A historical perspective on the next 20 years. *Energy Policy*, 50:109–116.

Sahin, C., Cayci, F., Gutiérrez, I. L. M., Clause, J., Kiamilev, F. E., Pollock, L. L., and Winbladh, K. (2012). Initial explorations on design pattern energy usage. In *First International Workshop on Green and Sustainable Software, GREENS '12, Zurich, Switzerland, June 3, 2012*, pages 55–61.

Sahin, C., Tornquist, P., Mckenna, R., Pearson, Z., and Clause, J. (2014). How does code obfuscation impact energy usage? In *IEEE International Conference on Software Maintenance and Evolution, ICSME '14*, pages 131–140.

Standard, R. (2013). GHG Protocol Product Life Cycle Accounting and Reporting Standard ICT Sector Guidance. In *Greenhouse Gas Protocol*, number January, chapter 7 - Guide.

Symantec (2008a). Corporate responsibility report. `http://www.symantec.com/content/en/us/about/media/SYM_CR_Report.pdf`. Accessed: 2014-09-23.

Symantec (2008b). Environmental progress and next steps. Via Email to Everyone Symantec (Employees). Internal email.

Vásquez, M. L., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Penta, M. D., and Poshyvanyk, D. (2014). Mining energy-greedy API usage patterns in android apps: an empirical study. In *11th Working Conference on Mining Software Repositories, MSR '14, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 2–11.

Vereecken, W., Van Heddeghem, W., Colle, D., Pickavet, M., and Demeester, P. (2010). Overall ict footprint and green communication technologies. In *4th International Symposium on Communications, Control and Signal Processing, ISCCSP '10*, pages 1–6.

Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '10, part of ESWeek '10 Sixth Embedded Systems Week, Scottsdale, AZ, USA, October 24-28, 2010*, pages 105–114.

Zhang, Y. and Ansari, N. (2013). Chapter 12 - Green Data Centers. In Obaidat, M. S., Anpalagan, A., and Woungang, I., editors, *Handbook of Green Information and Communication Systems*, pages 331–352. Academic Press.

Zheng, A. X., Jordan, M. I., Liblit, B., and Aiken, A. (2003). Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS '03, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*.

Appendices

A. Grammar used to define the input syntax of the results treatment phase

⟨*Input*⟩ ⟶ ⟨*Data*⟩*

⟨*Data*⟩ ⟶ ⟨*Component-begin*⟩ ⟨*Data*⟩* ⟨*Component-end*⟩

⟨*Component-begin*⟩ ⟶ '`>`' ⟨*Component*⟩

⟨*Component-end*⟩ ⟶ '`<`' ⟨*Component*⟩

⟨*Component*⟩ ⟶ ID '`[`' ⟨*Params*⟩ '`]`'

⟨*Params*⟩ ⟶ ⟨*Param*⟩ ('`,`' ⟨*Param*⟩)*

⟨*Param*⟩ ⟶ '`time`' '`=`' NUMBER
| '`cpu`' '`=`' NUMBER
| '`dram`' '`=`' NUMBER
| '`gpu`' '`=`' NUMBER
| '`fans`' '`=`' NUMBER
| '`disk`' '`=`' NUMBER

B. Grammar used to define the syntax of the input of the SPELL analysis phase

$$
\begin{aligned}
\langle \textit{Matrix} \rangle &\longrightarrow \langle \textit{Row} \rangle^* \\
\langle \textit{Row} \rangle &\longrightarrow \langle \textit{Component-Sample} \rangle^* \\
\langle \textit{Component-Sample} \rangle &\longrightarrow \text{`[' } \langle \textit{Params} \rangle \text{ `]'} \\
&\quad | \quad \text{`\_'} \\
\langle \textit{Params} \rangle &\longrightarrow \langle \textit{Param} \rangle \ (\text{`,' } \langle \textit{Param} \rangle)^* \\
\langle \textit{Param} \rangle &\longrightarrow \text{`time' `=' NUMBER} \\
&\quad | \quad \text{`numberUsed' `=' NUMBER} \\
&\quad | \quad \text{`cpu' `=' NUMBER} \\
&\quad | \quad \text{`dram' `=' NUMBER} \\
&\quad | \quad \text{`gpu' `=' NUMBER} \\
&\quad | \quad \text{`fans' `=' NUMBER} \\
&\quad | \quad \text{`disk' `=' NUMBER}
\end{aligned}
$$