



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Mário Nelson Araujo Santos

**Energy Analysis
in the CodeCompass system**

December 2017



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Mário Nelson Araujo Santos

**Energy Analysis
in the CodeCompass system**

Master dissertation

Master Degree in Computer Science

Dissertation supervised by

Prof. João Saraiva

Dr. Zoltan Porkolab

December 2017

ACKNOWLEDGEMENTS

First of all, I would like to express my deep gratitude to Professor João Saraiva for believing in my abilities, joining me early in the Green Software Lab (GSL) team and together with Dr. Zoltan Porkolab giving me the opportunity to make my thesis in the Erasmus+ Placement program, in Budapest and in a big company like Ericsson. In addition, I would particularly like to thank Dr. Zoltan for having received me in Budapest, getting me a scholarship and a good and cheap accommodation compared to the rest of students, and for having helped me in everything I needed to feel good and safe. In the same subject, I would also like to thank Professor Paulo Azevedo and the International Relations Services (SRI) of the University of Minho for having financed my Erasmus with an Erasmus+ Placement scholarship, without this huge help this partnership would be impossible.

I would particularly like to thank Dániel Kupp, one of Ericsson's members and one of the CodeCompass developers. He was the person who had the most enthusiasm and curiosity to help without any prior knowledge of energy efficiency or RAPL framework. Without his logical thinking and active participation, this project would not have gone so well. I would like to thank Dr. Zoltan, again, for helping me when I had doubts about Clang, for giving me the opportunity to submit an article to a conference (*Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA)*) and for going there to introduce it.

Lastly, I would like to thank Rui Pereira and Marco Couto, members of GSL, for making me aware of the work already done on Green Computing and the best ways to go during the development of this project.

ABSTRACT

Green computing has an increasing importance in software engineering. Unfortunately, there are lack of tools on this field to help developers to understand and fix issues related to unwanted energy consumption.

The thesis project will provide *Software (sw)* eng. with information about energy consumption of functions and methods. The CodeCompass system helps software developers to understand their source code, and it was developed by the Hungarian team members of Ericsson.

Thus, I will locate hot spots in the software's source code responsible for abnormal energy consumption, and I will do a plug-in to extend the CodeCompass tool so that it can automatically locate such energy faults, helping software developers to optimize the energy consumption of their software.

RESUMO

Computação verde tem uma importância crescente em engenharia de software. Infelizmente, há falta de ferramentas neste campo para ajudar os eng. de software a entender e corrigir problemas relacionados ao consumo de energia indesejados.

O projecto desta tese fornecerá aos desenvolvedores de software informações sobre o consumo de energia de funções e métodos. A ferramenta CodeCompass ajuda os engenheiros de software a entender o código-fonte e foi desenvolvida pelos membros da equipa húngara da Ericsson.

Por fim, localizarei zonas "quentes" no código fonte do software responsáveis pelo consumo anormal de energia e irei construir um plug-in para estender a ferramenta CodeCompass para que ela possa localizar automaticamente essas falhas de energia, ajudando os engenheiros de software na otimização do consumo de energia de seus programas.

CONTENTS

1	INTRODUCTION	1
1.1	Context and Motivation	1
1.2	Objectives	2
1.3	Research Group Context	3
2	BACKGROUND	5
2.1	Ericsson	5
2.1.1	Research and Development	5
2.2	Green Computing	6
2.2.1	Origins	6
2.2.2	Hardware	7
2.2.3	Software	7
2.2.4	Techniques for software development	8
2.3	State of the Art	9
2.3.1	Energy Aware Software Tools	9
2.3.2	Techniques for Green Software Analysis	13
3	ENERGY IN THE CODECOMPASS SYSTEM	21
3.1	Why does Ericsson needs Green Computing?	21
3.2	CodeCompass	22
3.2.1	Architecture	23
3.2.2	Web User Interface	25
3.2.3	Functionality	25
3.2.4	Perfomance	27
3.2.5	User Acceptance in Real Production	27
3.3	Green CodeCompass Plug-in	28
3.3.1	Decisions	28
3.3.2	Implementation	32
3.3.3	Plug-in Outcomes	39
4	CASE STUDIES / EXPERIMENTS	42
4.1	Experiment setup	42
4.1.1	Hardware Prerequisites	42
4.1.2	Software Prerequisites and Configuration	43
4.2	Results	45
4.2.1	TinyXml	45
4.2.2	Xerces-c-3.1.4	47

4.3	Discussion	48
4.3.1	Validating the Measurements	48
5	CONCLUSION	50
5.1	Conclusions	50
5.2	Prospect for future work	51

LIST OF FIGURES

Figure 1	Energy Star GHG (greenhouse gas) Reductions Since 2000	6
Figure 2	The behavior of the monitoring framework	10
Figure 3	Power domains for which power monitoring/control is available.	11
Figure 4	Sunflow: energy behaviors under different data precision choices	14
Figure 5	List results for population of 25k	15
Figure 6	Sometimes Faster doesn't mean Greener	16
Figure 7	The 13 Benchmarks available in <i>Computer Language Benchmark Game (CLBG)</i>	17
Figure 8	Normalized global results for Energy and Time	17
Figure 9	Language ranking considering all combinations of energy, time and memory (in Pereira et al. (2017b))	18
Figure 10	Generic formulation of a SFL Matrix	19
Figure 11	The Jaccard similarity coefficient	19
Figure 12	Static Model Formalization	19
Figure 13	SPELL embedded Gzoltar graphic visualization tool	20
Figure 14	CodeCompass architecture	23
Figure 15	User Interface	25
Figure 16	CodeBites example	26
Figure 17	CodeCompass usage distribution per task	28
Figure 18	System architecture	29
Figure 19	Visualization of Profiling Data in Kcachegrind (Weidendorfer and Zenith)	31
Figure 20	Example of an <i>Abstract Syntax Tree (AST)</i> with the nodes to be read.	33
Figure 21	How CRAPL works.	36
Figure 22	How energy measurement values are saved	37
Figure 23	All functions measurement with overhead (ppo)	41
Figure 24	All functions measurement without overhead (ppo)	41
Figure 25	x86 Intel Architectures that have RAPL interface(Weaver, 2015).	43
Figure 26	Tinyxml results without overhead	46
Figure 27	Tinyxml results with overhead	46

LIST OF TABLES

Table 1	Performance of CodeCompass v4	27
Table 2	Energy Consumption of functions executed in some tests	47
Table 3	Detailed energy consumption from DTest	48
Table 4	Validation of results in TinyXml (values represented in mJ)	49

ACRONYMS

A

AST Abstract Syntax Tree.

C

CLBG Computer Language Benchmark Game.

CSR Corporate social responsibility.

D

DGC Distributed Green Compiler.

DSL Domain-specific languages.

E

EPA Environmental Protection Agency.

F

FCT Fundação para a Ciência e Tecnologia.

G

GPU Graphics Processing Unit.

GREENSSCM Green Software for Space Control Mission.

GSL Green Software Lab.

I

ICSE International Conference on Software Engineering.

ICT Information and Communication Technology.

IDE Integrating Developing Environments.

L

LLVM Low Level Virtual Machine.

M

MSR Machine Specific Records.

R

RAPL Running Average Power Limit.

S

SFL Spetrum-based Fault Localization.

SPELL Spectrum-based Energy Leak Localization.

SQAMIA Software Quality Analysis, Monitoring, Improvement, and Applications.

SW Software.

INTRODUCTION

1.1 CONTEXT AND MOTIVATION

In the field of software engineering, recent studies have defined powerful techniques to increase software developers' productivity by supplying, for example, integrating developing environments *Integrating Developing Environments (IDE)*, testing and debugging frameworks and tools, advanced type and modular systems, etc. Furthermore, to improve the execution time of our software, compiler construction techniques were developed, namely by using partial and/or runtime compilation, parallel execution, advanced garbage collectors, etc. All of these engineering tools and techniques are designed to help software developers quickly specify correct programs with the ideal execution time. Unfortunately, none of these techniques or tools have been adapted to support green software development. In fact, there is no software engineering discipline that provides techniques and tools to help software developers understand, analyze, and optimize the power consumption of their software! As a consequence, if a developer realizes that his software is responsible for a large battery leak, he does not receive support from the language/compiler he is using.

In this thesis, I aim to study, develop, and apply methods to statically analyze abnormal energy consumption in software source code. Thus, the focus of the thesis is to reason about energy consumption at the software level more specifically in functions or methods of the given program written in the widely used C++ programming language. This is an innovative approach to analyze power consumption, since most of the research done on reducing the energy consumed by computers was done at the hardware level, not the software.

In this context, [Pereira et al. \(2017a\)](#) characterize an abnormal or excessive power consumption by a software system as an energy leak (inefficient energy consumption).

I would also like to evaluate the validity of these techniques in real-world, industrial-size software applications. The Ericsson research team in Budapest is involved in the develop-

ment of the CodeCompass framework. This framework is an extensible static analysis tool for analyzing and visualizing the program's source code written in different languages.

I will develop an energy-aware plug-in to locate abnormal energy consumption in the program's source code moreover, the located abnormal spots will be visualized and presented them to the software developers in the CodeCompass framework. Because CodeCompass is being used at Ericsson to analyze several of its software systems, I will validate my tool in the example systems where developers tested their plugins. Ericsson has just made CodeCompass open-source, and thus, the results of this thesis will be freely available for the green *sw* community and in the end, I plan to publish an article in top conferences and journals in the area of green software and sustainable computing, that have been created in recent years¹.

1.2 OBJECTIVES

While in the previous century software developers and computer manufacturers' primary goal was to produce very fast software systems and computers, in this century this has changed: the widespread use of non wired but powerful computer devices is making battery consumption/lifetime the bottleneck for both manufacturers and software developers.

The hardware manufacturers are already aware of this concern and a lot of work in terms of optimizing energy consumption by optimizing the hardware has being done. Unfortunately, the programming language and software engineering communities have not fully understood this bottleneck, and as a result there is little support for software developers to discuss the power consumption of their software. Although the hardware is the one that consumes energy, the software can greatly influence this consumption, such like a driver who operates a car influences its fuel consumption.

Furthermore, I will extend the CodeCompass tool to identify such programming factors in the source code of software systems. This tool uses static analysis techniques to analyze programs written in C/C++, Java and Python. It computes multiple software metrics and presents an user-friendly visualization of such metrics and source code. In this thesis, I will implement an energy conscious plug-in for CodeCompass so that those identified factors are presented to the software developers in the source code they are developing in C or C++, the main two programming languages that Ericsson Budapest usually works on their projects and test on this tool. In the end, I plan to use industrial-size software systems from

¹ See for example the GREENS workshop - <http://greens.cs.vu.nl/> - that is now part of the top conf. on *sw* engineering *International Conference on Software Engineering (ICSE)*

Ericsson Budapest as case examples to test the plug-in. This thesis aims at answering the following three research questions:

- RQ1: Can we instrument a all C or C++ software system to add the *Running Average Power Limit (RAPL)* interface without compromise the execution of the programme?
- RQ2: Can we measure the energy of all the functions/methods of a project to easily check which ones are wasting more energy?
- RQ3: Can such techniques be implemented as a plugin of the CodeCompass tool? How efficient and effective are such techniques when handling industrial-size *sw* applications?

The results of this thesis will allow programmers to become energy aware during programming and with the appropriate tools they finally have ways to support green decision making.

1.3 RESEARCH GROUP CONTEXT

The University of Minho and the Faculty of Informatics of Eötvös Loránd University, Budapest have a history of scientific cooperation in the framework of the Erasmus project since 2007. The cooperation consists in students exchange, guest lectures and cooperation on teaching at summer schools. University of Minho has strong experiences on energy aware computing. In 2015, the coordinator of this thesis presented his ongoing research work on the analysis of software energy consumption in an invited talk at Ericsson in Budapest. The feedback of this talk was very positive and they discussed possible collaborations by implementing static analysis techniques in Ericsson software. So, in this context of Minho/Etvos Erasmus agreement, I was granted an Erasmus Placement Scholarship to do part of my MSc work at Ericsson, co-supervised by Prof. Zoltan Porkolab.

The static analysis research group at Eötvös Loránd University has been working on various static analysis tools for industrial applications for the last five years. Among other achievements, they implemented CodeChecker, an open source static analysis framework for the *Low Level Virtual Machine (LLVM)*²/Clang³ compiler infrastructure to detect programming anti-patterns, code smells and other issues applying both the analysis of the *AST* and using symbolic execution. Another achievement is CodeCompass – a code comprehension tool for C, C++, Java and Python languages. With CodeCompass, the developers are able to locate, browse and visualize large code bases where a certain feature is implemented. The

² LLVM is a collection of modular and reusable compiler and toolchain technologies.

³ Clang is a compiler front end for various programming languages. It uses LLVM as its back end.

plugin-able structure of the CodeCompass tool makes it possible to extend the framework with new modules, in our case the visualization of energy aware information among the source code. The role of the Hungarian partner is to define the best methods to present the information about energy leaks and other issues in the system and extend CodeCompass with the necessary parsers and plug-ins.

Four years ago, the researchers of Minho started the *Green Software Lab (GSL)* to study and develop techniques and tools for green software. In the scholar year 2013/14, João Saraiva included the study of techniques for green software in the MSc and PhD courses that he teaches at the MSc in Informatics Engineering and the MAPi doctoral program. This hot topic already attracted several young researchers, like Rui Pereira and Marco Couto.

The *GSL* is also actively applying the research in industrial settings. Last year the team started applying energy consumption estimation techniques to aerospace software in the context of the project *Green Software for Space Control Mission (GreenSSCM)*. Moreover, the team is also active applying for research project funding. Prof. João Saraiva is the principal investigator of a project on green software funded by *Fundação para a Ciência e Tecnologia (FCT)*. He is also the coordinator of a *FCT/Slovakia* bilateral project⁴ on green computing (2016-2017).

⁴ <https://kpi.fe.i.tuke.sk/en/user/szabo-csaba/towards-a-software-engineering-discipline-green-software>

BACKGROUND

In this chapter I will present a study about the background of the project. Firstly, I will describe in detail the main topic of this thesis, namely **Green Computing**. In general, I will talk about the history, why it was created such concept and the importance it has in the *Information and Communication Technology (ICT)* sector.

Then I will introduce technologies\innovations to reduce energy consumption, both hardware and software level. Finally, I start in the next section by introducing Ericsson where the main part of the MSc thesis was conducted.

2.1 ERICSSON

Founded in 1876 by Lars Magnus Ericsson, Ericsson is one of the world's largest telecommunication network and equipment companies. Headquartered in Sweden, the company offers various services such as software and infrastructures in information and communication technology, traditional telecommunications and Internet Protocol (IP), mobile and fixed broadband, cable television, video systems, and operations and business support services. Worldwide, it operates in about 180 countries and it employs around 98.000 people, according to [Ericsson \(2016\)](#).

2.1.1 *Research and Development*

The research and development team is part of Group Function Technology, which includes several universities and research institutions, such as: Lund University (Sweden), Eötvös Loránd University (Hungary) and Beijing Institute of Technology (China). Group Function Technology focuses primarily on wireless access networks, broadband technologies, packet technologies, multimedia technologies, radio access technologies, software, security and global services. This dissertation will be done under the context of the **CodeCompass** comprehension tool, one of the projects developed between **Ericsson** and **Eötvös Loránd University**, which aims to help programmers understand large-scale software systems from

mainly static analysis of code.

2.2 GREEN COMPUTING

While in the early days of computing, when developing hardware\software the primary goal was to get systems to run as fast as possible, in recent years we have seen a significant increase in research of the development and production of hardware and software components with low levels of energy consumption. Sustainability is currently one of the world's most important issues, making energy consumption one of the most critical headaches in ICT.

2.2.1 Origins

Although the sustainability of our planet starts several decades ago, the first steps of Green Computing came in 1992 with the appearance of the Energy Star (ENERGY (2011)), a voluntary certification program created by the United States *Environmental Protection Agency (EPA)*. The Energy Star was designed to identify and promote energy-efficient products so anyone could save money spent on light bills and reduce greenhouse gas emissions (Figure 1). Monitors, temperature control equipment and television sets were the first products to receive certification.

GHG REDUCTIONS (MMTCO ₂ e)														
GHGS ADDRESSED: CO ₂														
KEY SECTORS: Residential, Commercial, Industrial														
2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	1992-2013 CUMULATIVE
53.5	64.9	78.1	91.7	103.8	115.5	128.3	144.8	156.2	169.8	195.8	221.2	254.7	293.9	2,198

Figure 1: Energy Star GHG (greenhouse gas) Reductions Since 2000

According to Scheild (2011), the first important, yet very simple result of Green Computing was the creation of the suspend mode function on computer monitors. Over the years, the concept has been developed from other solutions, such as thin client¹, cost accounting, virtualization practices and eWaste (recycling of electronic products).

¹ A thin client is a lightweight computer that is purpose-built for remote access to a server.

2.2.2 Hardware

According to [Calero and Piattini \(2015\)](#), one of the important steps for sustainability passes through the day-to-day of the individual person. Learning how to recycle used materials and reducing carbon dioxide emissions in our routine (using energy-efficient computers or using public transports more often, for example).

At hardware level there are some curious ideas\technologies ([Statham et al. \(2012\)](#)):

- **The RITI printer** - printer that uses coffee grounds as a substitute for ink;
- **Green hard drives** - they use less energy when running and conserve energy when not being used;
- **Solar Computing** - Intel has developed a low-power processor capable of running on solar-cell PCs based on solar cell technology, i.e, a seal-sized electronic component that converts sunlight into electricity.

2.2.3 Software

[Bener et al. \(2014\)](#) said that most of the spending on Green ICT comes from the effects of hardware on the environment. People have little consideration for the impact of software products. Although software systems do not directly consume power, they affect the use of hardware components.

Efficiency in energy consumption, efficient allocation of space, efficiency in memory capacity, usability, availability and storage of data and information, and efficiency in the use of planning time (parallelism) are some aspects that may be important when developing energetically efficient Software. In fact there are several technologies that can significantly reduce energy consumption of *sw* systems, namely:

- **Virtualization** - directly reduces the use of hardware required for multiple systems. On a server or desktop, virtualization allows multiple operating systems or applications to run on a single computer - [Turban et al. \(2008\)](#);
- **Terminal servers** - users connect to a central server and all real computing is done on the server. They can be combined with thin clients that only use 1/8 the amount of power of a normal workstation - [Bener et al. \(2014\)](#);
- **Power management** - allows the operating system to control the power management features connected to the hardware. The system can automatically shut down components such as hard drives or the monitor if they are inactive after a certain period of time - [Star](#);

- **Cloud computing** - according to Mines:

1. Automation software, maximizing consolidation and utilization to drive efficiencies;
2. Pay-per-use and self-service, encouraging more efficient behavior and life-cycle management;
3. Multi-tenancy², delivering efficiencies of scale to benefit many organizations or business units.

2.2.4 Techniques for software development

During the software development cycle energy consumption can be reduced, starting with the analysis up to design and implementation. In the design we can take into account some energy-efficient structures for the project. In the implementation there are some parameters that we can take into account, according to [Fakhar et al. \(2012\)](#):

1. **Use of green IDE & compiler** - use of compilers capable of reshaping the source code by applying various Green optimizations during code transformation. Green Hill compiler for C and C++, encc energy aware compiler for C++, are some examples.
2. **Recursion vs. Iteration** - Recursion uses stacks. At the beginning of each function, the arguments have to be pushed in the stack and at the end of the function they have to be popped, which leads to more execution time, thus also leading to more power consumption. Therefore, we should avoid recursion and use iterations. Recursion elimination is a key optimization automatically performed by compiler.
3. **Less running time** - Normally, any reduction in run time may be useful to reduce energy consumption. Therefore, always try to use algorithms with linear complexity.
4. **Use of energy aware data structure** - data structures have a significant effect on the conservation of energy and in the execution of a program. For example a study made by [Couto et al. \(2017\)](#), in a merge sort an array of arrays consumes less energy compared to a link list.
5. **Algorithmic efficiency** - programmers must write efficient algorithms by writing a code-specific design and data structures based on the application, programming language and hardware architecture.
6. **Sacrificing performance above a limit for energy efficiency** - sometimes it is better to be greener than faster.

² Multi-tenancy is an architecture in which a single instance of a software application serves multiple customers.

7. **Code written for energy allocation purposes** - you can route traffic to locations with lower energy costs or less hot climates.
8. **Define IT Resource and Quality Metrics** - in the analysis phase it may be important to define some metrics to measure (Mahmoud and Ahmad, 2013):
 - Total life-cycle costs of the process - take into account parameters such as programmer experience, complexity of the operation, integration and reuse rates, and required stability;
 - Power consumption and efficiency - measure the amount of power supply, materials consumed, CO₂ emissions and other energy-related factors released by the air.
 - Infrastructure costs, human efforts, material outputs and compliance with environmental laws.

2.3 STATE OF THE ART

In this section I will present studies conducted by some researchers. Some techniques and measures concluded by them that can significantly reduce energy consumption when developing a certain application, as well as tools designed to help developers to save energy or to reach certain conclusions in their studies on energy efficiency.

2.3.1 *Energy Aware Software Tools*

Software can also play an active role in saving energy by providing feedback on how software components consume resources, thereby enabling programmers to create greener processes. Here are some examples:

Distributed Green Compiler (DGC)

DGC reshape the source code during intermediate code conversion by applying various green techniques to produce an energy efficient executable program. A green compiler requires more time to compile the source code compared to normal compilers. It applies ecological strategies in the compilation.

According to Fakhar et al. (2012), it also provides green suggestions to programmers, highlighting areas of source code that can not be transformed by the compiler for energy optimization during intermediate code conversion. DGC gives program energy consumption statistics after compilation, telling the programmer how much energy can be saved in a produced executable. Loop optimization, use of energy optimized data structure, dead

code elimination, software pipelining, recursion elimination, cloud aware task mapping, un-optimized code blocks identification and energy cost statistics are some of the suggestions\transformations that the compiler offers.

GreenDroid

GreenDroid is a framework that profiles the energy consumption on the Android ecosystem (Couto et al., 2014). The tool can be used to determine the methods of an Android application that are likely to be associated with abnormal energy consumption. Greendroid uses the Android sw test framework to execute the program and monitor the energy consumed by the executed methods. Once the source code and application tests are ready, the tool performs a set of sequential steps to present the desired results.

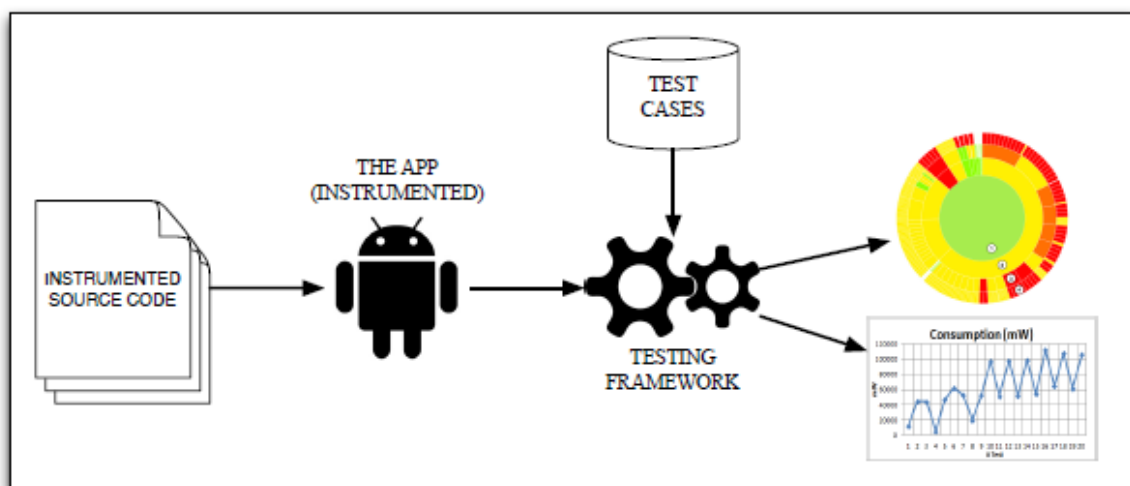


Figure 2: The behavior of the monitoring framework

As shown in Figure 2, the steps are:

1. **Execute the tests** - the tests run twice, once to get the list of methods invoked and another to measure power consumption. The result will be stored in files containing a list of the invoked methods, as well as the number of times it was invoked, the test execution time and the energy consumed, in mW (Milliwatts).
2. **Merge the results** - after all tests have been executed, the tool will generate a set of files corresponding to the number of tests. Then the they will be gathered into a single file to be read and analyzed to extract the information.
3. **Classify the methods** - in this step, the tool will read the file values and sort them according to green-aware metric values.

4. **Generate the results** - in the end it will generate graphical representations of the components of the source code, giving it different colors according to the classification specified in the previous step.

RAPL

In this section I overview how one can measure the energy consumption of a processor using the Intel's [RAPL](#) interface.

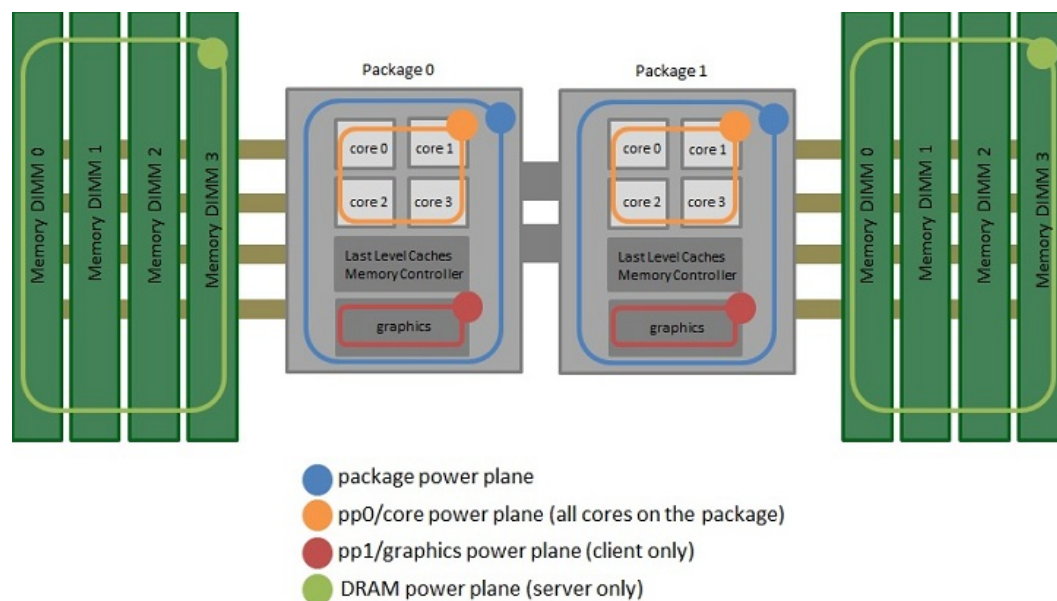


Figure 3: Power domains for which power monitoring/control is available.

According to [Dimitrov et al. \(2015\)](#), [RAPL](#) was designed by Intel as a set of low-level interfaces with the ability to monitor, control, and get notifications of energy consumption of different hardware levels. It is supported in today's Intel architectures, like i5 and i7 CPUs. The architectures, that support [RAPL](#), monitor energy consumption information and store it in *Machine Specific Records (MSR)*³. These *MSR* can be accessed by the Operating System.

As shown in [Figure 3](#), [RAPL](#) allows energy consumption to be reported in a practical way, by monitoring:

- **Package (PKG):** entire socket (pp0+pp1);
- **Power Plane 0 (PP0):** all of the CPU cores in the package;

³ A model-specific register (MSR) is any of various control registers in the x86 instruction set used for debugging, program execution tracing, computer performance monitoring, and toggling certain CPU features.

- **Power Plane 1 (PP1):** uncore. Often provides info for the integrated *Graphics Processing Unit (GPU)*;
- **DRAM:** DRAM in the system.

Rapl Interfaces

RAPL interfaces consist of non-architectural MSR. According to Intel (64) the following set of capabilities are supported by each RAPL domain:

- **Power limit** - MSR interfaces to specify power limit and time window;
- **Energy Status** - Power metering interface providing energy consumption information;
- **(Optional) Perf Status** - Interface providing information on the performance effect due to power limits;
- **(Optional) Power Info** - It is an interface that provides information on the set of parameters for a given domain, maximum power, minimum power, etc;
- **(Optional) Policy** - 4-bit priority information that is a hardware tip for dividing the budget between sub-domains in a parent domain.

Each of the above features needs specific units to describe them. Time is expressed in seconds, power is expressed in Watts and energy is expressed in joules. The scaling factors are provided to each unit to make the information presented significantly in a finite number of bits. Units for power, energy, and time are exposed in the read-only MSR_RAPL_POWER_UNIT MSR.

There are several implementations/libraries in different programming languages to access RAPL measurements. Next I briefly show the Java binding of RAPL, called jRAPL.

jRAPL

According to jRA, jRAPL is a framework for profiling Java programs running on CPUs with RAPL support.

```
double beginning = EnergyCheck.statCheck();
doWork();
double end = EnergyCheck.statCheck();
```

As one can see with the example above, it can be viewed as a software wrapper to access the MSR.

Trepm Power Profiler

A [RAPL](#) equivalent for mobile devices, developed by Qualcomm Technologies. ([Qualcomm Technologies](#)) [Trepm™ Profiler](#) is an on-target power and performance profiling application for mobile devices. With this tool, programmers can understand the impact of their programming choices on both performance and energy. Some features provided by Trepm:

- Six fast-loading profiling presets;
- Overlays appear on screen on top of applications that are being profiled;
- Profile a single app, or your device;
- Displays battery power on supported devices;
- View CPU and [GPU](#) utilization and frequency;
- Display network usage (Wi-Fi and cellular);
- Advanced mode to manually select data points and save data for later analysis.

SEEP

Developed by [Hönig et al. \(2011\)](#), SEEP is designed to aid program development by analyzing source code and obtaining accurate platform-dependent power profiles, per function, using pre-existing knowledge about the energy consumption of the underlying instructions. The source code must be compiled into an intermediate representation and provided as input to the SEEP program in order to obtain a high degree of code coverage and therefore high precision. Subsequent results need to be combined with code models in the source code and the process repeated in case of updates to the code base.

2.3.2 Techniques for Green Software Analysis

This section presents studies about energy efficiency in programming. It will be shown some green rankings and some techniques to understand where power failures (or energy leaks) are located, in which component a program may be spending a lot of energy. Most of these studies were conducted by members of the GSL group where this thesis is integrated.

Data-Oriented Characterization of Application-Level Energy Optimization

To show how jRapl works, Liu et al. (2015) used Sunflow⁴. The program represents rendering data in type double, so they changed and tested the program using other primitive data types such as short, int, float and long.

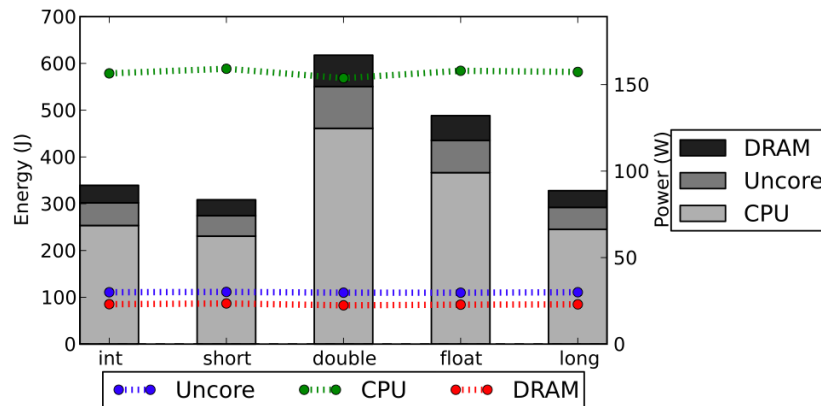


Figure 4: Sunflow: energy behaviors under different data precision choices

As shown in Figure 4, the short type is the one that uses less energy compared to the rest.

Although Sunflow is a complex application, with more than 20,000 lines of code we can notice that a simple modification in the usage of data types in a method can have a big impact on the energy consumption of the application.

Ranking of Java Data Structures

Researchers have studied the energy consumption of the Java Collections Framework Pereira et al. (2016). In this study, the authors considered all of the collections within the framework and grouped each collection by their implemented interface, as shown in the following list:

- JCF Data structures:

Sets ConcurrentSkipListSet, CopyOnWriteArraySet, HashSet, LinkedHashSet, TreeSet

Lists ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

Maps ConcurrentHashMap, ConcurrentSkipListMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, Properties, SimpleBindings, TreeMap, UIDefaults, WeakHashMap

⁴ Sunflow renders a set of images using ray tracing, a CPU-intensive benchmark.

Using varying population sizes of 25,000, 250,000, and 1,000,00 elements, they measured the energy consumption, by using jRAPL jRA, of each method within their specific API list. Below is the complete list of the analyzed methods.

- Methods:

Sets add, addAll, clear, contains, containsAll, iterateAll, iterator, remove, removeAll, retainAll, toArray

Lists add, addAll, add (at an index), addAll (at an index), clear, contains, containsAll, get, indexOf, iterator, lastIndexOf, listIterator, listIterator (at an index), remove, removeAll, remove (at an index), retainAll, set, sublist, and toArray

Maps clear, containsKey, containsValue, entrySet, get, iterateAll, keySet, put, putAll, remove, and values

By applying this design, the authors were able to achieve an energy efficiency ranking of the collections based off of each method’s energy consumption. Additionally, these rankings are grouped by different combinations of the implemented interfaces, *Sets*, *Lists*, and *Maps*, and the different population sizes. All of this allows a developer to have a better understanding of which data structure would be best suited for a given scenario. Below, in Figure 5, is an example of one of the data tables for the *List* collections using a population of 25,000.

Methods	ArrayList		AttributeList		CopyOn Write ArrayList		LinkedList		RoleList		Role Unresolved List		Stack		Vector	
	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms
add	0.9773	71	1.1510	67	1.7839	117	1.8016	86	1.4801	76	1.1865	74	1.5659	76	1.5177	69
addAll	1.3353	76	1.0492	88	1.3586	82	1.1043	88	1.6661	76	1.8672	88	1.1015	88	1.7903	73
addAlli	1.7855	86	1.6035	68	1.1789	86	1.7272	99	1.5980	81	1.2497	85	1.2962	72	1.6268	90
addl	1.7125	93	1.3849	87	1.6558	119	1.6404	96	1.2718	85	1.3124	86	1.5287	83	1.4554	86
clear	1.1284	76	1.2409	75	1.7155	68	1.6497	74	1.6705	76	1.4304	80	1.6199	73	1.0574	71
contains	2.7568	166	2.4228	165	3.1768	167	3.1552	193	2.1751	162	2.4688	164	2.0128	166	2.1558	168
containsAll	1.5993	87	1.8053	92	2.1889	92	2.2887	118	1.3244	100	1.3930	96	1.2054	89	1.5091	87
get	2.0029	83	1.1171	78	1.4918	77	2.0168	109	2.2110	81	1.6613	71	1.8956	86	1.4978	73
indexOf	1.4447	76	2.0325	84	1.5682	70	2.6289	101	1.5674	79	1.1944	81	1.8090	81	2.0788	75
iterateAll	2.0701	79	1.0473	77	1.0103	73	2.6401	107	1.3605	85	1.7822	71	1.6036	81	1.1336	87
iterator	1.4893	84	1.1589	84	1.3922	72	1.7666	108	1.9760	73	1.3300	79	2.1895	84	1.6505	83
lastIndexOf	1.7750	99	1.7666	98	2.0383	94	2.5019	127	1.8914	92	1.4211	95	1.2260	84	1.2296	96
listIterator	1.4457	76	1.6190	84	1.3737	71	2.5003	106	1.3380	80	1.5176	85	1.6354	69	1.2746	81
listIteratori	1.7356	78	1.1552	81	1.5160	77	2.1996	105	1.7588	79	1.0334	80	1.8799	85	1.7545	78
remove	1.1308	96	1.4480	85	2.1946	162	1.6924	98	1.4560	84	1.1368	85	1.2663	96	1.4973	82
removeAll	8.0905	671	7.8108	697	7.3237	666	8.3150	752	7.6148	692	7.9911	664	7.3824	654	7.1281	665
removei	1.9135	85	1.3534	92	2.2858	118	1.7174	100	1.6308	85	1.6369	89	1.5850	81	1.5486	90
retainAll	2.7037	193	2.7845	200	2.6052	198	2.5982	205	3.0973	197	2.4172	200	2.7635	242	3.4019	245
set	0.9476	64	1.5943	70	1.9669	110	2.0474	112	1.5249	76	1.2312	73	1.4938	75	1.4957	72
sublist	1.3108	76	1.6021	80	1.4792	80	1.8457	98	1.4910	85	1.5117	71	1.7082	75	0.9414	75
toArray	1.6418	84	1.5024	84	2.0934	73	1.6739	106	1.5418	79	1.7455	83	1.5694	69	2.0213	80

Figure 5: List results for population of 25k

From the data presented in Figure 5, for *Lists* with a size of 25,000, we can see that RoleUnresolvedList and AttributeList are the collections which have the tendency to consume the least amount of energy, while LinkedList is the least efficient implementation. Additionally, the energy measurements are colored with a varying spectrum of colors from red to green, depicting a method to be inefficient or efficient respectively.

Methods	Properties		Simple Bindings	
	J	ms	J	ms
clear	2.0777	98	2.1401	106
containsKey	1.6018	107	1.8055	99
containsValue	7.3755	692	7.9912	678
entrySet	1.7800	97	2.1557	102
get	1.7851	97	1.5359	100
iterateAll	1.6362	100	2.0472	116
keySet	1.4866	92	2.0630	106
put	1.7038	107	2.1646	102
putAll	1.3097	95	2.1461	112
remove	1.9660	98	2.2178	106
values	1.9120	111	2.0692	108

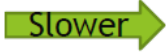
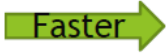



Figure 6: Sometimes Faster doesn't mean Greener

This study also supports the statement that a faster implementation does not always mean a more energy efficient one. As shown in Figure 6, the `containsKey` method in `Properties` is slower than in `SimpleBindings`, yet it consumes less energy.

Finally, using a very simple methodology to choose a more efficient collection, the authors were able to optimize Java projects by an average of 6.2%.

In another similar study to this one, [Lima et al. \(2016\)](#) have studied the energy consumption of different data structures in **Haskell**.

Towards a Green Ranking for Programming Languages

In order to better understand how the choosing of a particular programming languages can influence the energy consumption of a software solution, researchers have conducted studies that use implementations of the same problems in different languages aim to compare them. An example of such study is the one conducted by [Couto et al. \(2017\)](#): the authors used a set of computational problems, written in several languages, to check which of the considered programming language are more energy efficient.

The case study considered for the study was obtained from the repository available in the **CLBG** project. This repository contains solutions for 13 different problems, each one implemented in almost 27 programming languages. Figure 7 presents a brief description of the 13 **CLBG** problems.

This study compared 10 of the 27 languages: *C*, *Java*, *Octran*, *Fortran*, *C#*, *Go*, *Racket*, *Moon*, *Jruby*, an *Perl*, and only the first 10 problems were considered in this study, since there was

Benchmark	Description	Input
n-body	Double precision N-body simulation	50M
fannkuch-redux	Indexed access to tiny integer sequence	12
spectral norm	Eigenvalue using the power method	5,500
mandelbrot	Generate Mandelbrot set portable bitmap file	16,000
pidigits	Streaming arbitrary precision arithmetic	10,000
regex redux	Match DNA 8mers and substitute magic patterns	fasta output
fasta	Generate and write random DNA sequences	25M
k nucleotide	Hashtable update and k-nucleotide strings	fasta output
reverse complement	Read DNA sequences, write their reverse-complement	fasta output
binary trees	Allocate, traverse and deallocate many binary trees	21
chameneos redux	Symmetrical thread rendezvous requests	6M
meteor contest	Search for solutions to shape packing puzzle	2,098
thread ring	Switch from thread to thread passing one token	50M

Figure 7: The 13 Benchmarks available in CLBG

no solution of the last 3 in all the languages they wanted to test. The results of this study are presented in Figure 8.

Total			
	Energy	Time	
C	1.00	1.00	C
Java	1.68	1.65	Java
Ocaml \downarrow_1	2.13	2.44	C# \downarrow_2
Fortran \downarrow_2	2.20	2.48	Ocaml \uparrow_1
C# \uparrow_2	2.21	2.63	Go \downarrow_1
Go \uparrow_1	3.04	3.91	Fortran \uparrow_2
Racket	7.35	10.29	Racket
Lua \downarrow_2	39.54	44.68	Jruby \downarrow_1
Jruby \uparrow_1	45.35	68.45	Perl \downarrow_1
Perl \uparrow_1	84.89	77.10	Lua \uparrow_2

Figure 8: Normalized global results for Energy and Time

As it can be verified, it was concluded that the C language is the fastest and the most energy efficient in relation to all the others, being in second Java that energetically costs 1.69x more compared to C. They also concluded that not always fast means energetically efficient.

This study was extended afterwards [Pereira et al. \(2017b\)](#), in order to include the remaining 17 languages and to offer a more thorough comparison between the languages. While the first study focused mainly on comparing the energy consumption with the execution time in and between languages, the second study also presented a discussion about the memory consumption and its relation with the solutions' energy consumption, and a comparison considering not only languages but also programming paradigms.

Moreover, the authors verified that if they tried to rank the languages according to energy consumption, execution time and memory usage of the solutions, the rankings would be different. Given that, they aimed at creating a general language ranking, considering the 3 referred factors, and proposed such ranking as one of the main contributions of their study. The resulting ranking is presented in Figure 9.

Time & Memory	Energy & Time	Energy & Memory	Energy & Time & Memory
C • Pascal • Go	C	C • Pascal	C • Pascal • Go
Rust • C++ • Fortran	Rust	Rust • C++ • Fortran • Go	Rust • C++ • Fortran
Ada	C++	Ada	Ada
Java • Chapel • Lisp • Ocaml	Ada	Java • Chapel • Lisp	Java • Chapel • Lisp • Ocaml
Haskell • C#	Java	OCaml • Swift • Haskell	Swift • Haskell • C#
Swift • PHP	Pascal • Chapel	C# • PHP	Dart • F# • Racket • Hack • PHP
F# • Racket • Hack • Python	Lisp • Ocaml • Go	Dart • F# • Racket • Hack • Python	JavaScript • Ruby • Python
JavaScript • Ruby	Fortran • Haskell • C#	JavaScript • Ruby	TypeScript • Erlang
Dart • TypeScript • Erlang	Swift	TypeScript	Lua • JRuby • Perl
JRuby • Perl	Dart • F#	Erlang • Lua • Perl	
Lua	JavaScript	JRuby	
	Racket		
	TypeScript • Hack		
	PHP		
	Erlang		
	Lua • JRuby		
	Ruby		

Figure 9: Language ranking considering all combinations of energy, time and memory (in [Pereira et al. \(2017b\)](#))

Spectrum-based Energy Leak Localization (SPELL)

SPELL is a language independent technique to detect energy inefficient fragments in the source code of a software system. It is based off of *Spectrum-based Fault Localization (SFL)*, a statistical analysis technique typically used to locate bugs or program faults based off of several test executions.

More specifically, **SFL** uses a set of flags which reflect whether or not a concrete component is used in a particular execution of the software and set up an $n \times m$ (Figure 10) matrix (m different components and n different executed tests cases).

Each position of the matrix (a_{ij}) can either have the value of 0, if the component was not executed or the value 1 if it was. Finally, an error vector is constructed to state if an error occurred (1), or not (0) during a specific test execution.

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

Figure 10: Generic formulation of a SFL Matrix

$$J = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}.$$

Figure 11: The Jaccard similarity coefficient

Using the Jaccard similarity coefficient (as shown in Figure 11), SFL calculates how probable a certain component contains a fault. The variables of the coefficient function are as follows:

M₁₁ represents the total number of attributes where A and B both have a value of 1.

M₀₁ represents the total number of attributes where the attribute of A is 0 and the attribute of B is 1.

M₁₀ represents the total number of attributes where the attribute of A is 1 and the attribute of B is 0.

M₀₀ represents the total number of attributes where A and B both have a value of 0.

As one can not relate energy consumption to simple binary values of high (1) or low (0) consumption, the authors of SPELL adapted the SFL technique to allow it to be used for energy leak localization.

First they adapted the hit spectrum to a static energy model. Similar to the prior technique, it also uses an n x m matrix. But unlike SFL, the elements of the matrix are triples, as shown in the Figure 12:

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

Figure 12: Static Model Formalization

- E - for energy consumption (Joules);
- N - for the number of executions (cardinality);
- T - for runtime (milliseconds).

Finally, using an adapted similarity function, the technique uses a statistical method to relate energy consumption to different different source code components of a system, thus directing the developer's attention on the most critical "red" points in his code (as shown in Figure 13).

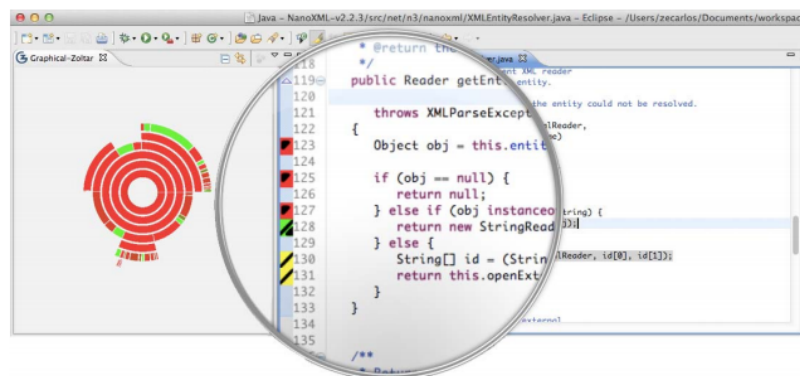


Figure 13: SPELL embedded Gzoltar graphic visualization tool

The main difference between SPELL and other static approaches is that instead of using a binary decision, the error vector is calculated by this technique giving it two different perspectives, Component Category Similarity and Global Similarity, to calculate the oracle and similarity. The first one is an analysis in a specific category (for example, considering only the energy consumption) and the second one is a global analysis considering the three.

Energy Consumption in Data Warehouses

In the database systems field, where the computational resources required during the processing of large volumes of data are enormous, caused the need to look for new implementations arose and from a study carried out by [Guimarães et al. \(2016\)](#) a new technique was created so that it was possible to categorize and evaluate the energy consumption in data warehouse settlement systems and thus to have the necessary information so that it is possible to build new settlement systems with lower energy costs. Very closely, the implementation consists in evaluating the energy consumption of all the components used in the settlement process from a conventional tool.

ENERGY IN THE CODECOMPASS SYSTEM

This chapter presents the design and the implementation of the Green CodeCompass plug-in. First, I will explain why companies like Ericsson need to be aware about energy efficiency. I will show the environment where this dissertation is developed. I will describe in detail about the tool where I will implement the green plug-in I propose and at the same time I will analyze the importance of it for large-scale projects in Ericsson's development teams. Then I will explain about the decisions that I had to make to be possible the measurement of the energy consumption of the industrial projects. I will also explain in detail the architecture of the system and the implementation of all the essential components of the whole tool. Finally, I will show an example of what the tool generates after running on a project.

3.1 WHY DOES ERICSSON NEEDS GREEN COMPUTING?

According to [Calero and Piattini \(2015\)](#), a company can receive huge public criticism and subsequently lose market legitimacy if it does not have sustainable in its top priorities. 47% of institutions began modernizing their sustainability-based business models, conducting sustainable development as a new source of innovation, a new mechanism to gain competitive advantage and a new opportunity to **cut costs**. Most people claim that they will pay more for an environmentally friendly product.

The ISO 26000 standard for *Corporate social responsibility (CSR)* was published, in early 2010. It provides executive guidelines and measures to demonstrate **social responsibilities**. The goal of this standard is to encourage the adoption of environmentally friendly information technologies and to promote greater environmental responsibility through business practices. Companies should take a precautionary approach to protect the environment.

For example, according to [Alves et al. \(2012\)](#), the environmental impact of datacenters (clusters, grids or clouds) is enormous, and studies have found that carbon dioxide (CO₂) emissions from multiple data centers are bigger than many countries. Therefore, it is possible to affirm that the power consumption of these large-scale and distributed equipment is enormous. With a huge energy consumption, another problem is also associated with

the companies that use these structures: the price to be paid for the high amount of energy consumed, also counting on the cooling and communication systems associated with this system. "Close to 50% of the **energy costs** of an organization can be attributed to the IT departments" - [Harmon and Auseklis \(2009\)](#).

In addition to organizations, programmers themselves are also beginning to be concerned about the energy efficiency of their programs. This conclusion was based on a study carried out by [Pinto et al. \(2014\)](#) and using StackOverflow as the main source, where they analyzed 300 questions and 550 answers from more than 800 users, related to the energy consumption of *sw*.

3.2 CODECOMPASS

The maintenance of large-scale software has always been problematic. Over the years, the design structure becomes fickle as developers change, the code becomes difficult to understand and the documentation tends to become unstable. To reduce these problems that are constant in projects coming from its work teams, Ericsson developed **CodeCompass Cod** ([2016](#)), a software comprehension tool for C/C++, Java and Python based on LLVM\Clang. The open-source tool was built to make it easier to understand that type of software.

From several studies on Ericsson projects these were the main requirements to build a good understanding tool:

- **Growing complexity** - Projects of this kind are constantly growing and becoming more complex, i.e the cost of fixing an error or adding new functionality also grows. Regardless of its complexity, project analysis must be scalable.
- **Knowledge erosion** - In a multinational enterprise environment, such as Ericsson, switching the developers of a large-scale software development teams is frequent, and the new programmers need to quickly adapt to the project.
- **Multiple views of the software based on various information sources** - In order to have a complete analysis of the program, different types of analysis must be carried out, from the compilation of the program to the analysis of metrics. Each one has its importance but it is necessary to combine all the possible analyzes in a single work environment.
- **Communication barriers** - Development teams that are located in different offices tend to have communication problems. If there is some kind of error, the teams will eventually argue on which side that error appeared. This is normally due to the fact that they do not know the intended behavior of the components of each other. An understanding tool should improve sharing of knowledge about components and teamwork.

- **Multiple programming languages** - Usually large-scale software systems are implemented in more than one language. It is necessary that the tool supports several languages in the same interface so that it is possible to navigate between the modules and making the usability of the tool more user-friendly.
- **Hard to deploy new tools** - It is often difficult to convince developers to use new tools, especially when they are hard to install or do not have a user-friendly interface. For this, it is necessary that the tool is intuitive and easy to install and use.
- **Requirement of open extensibility** - When planning a long-term software product *Domain-specific languages (DSL)* are used to describe the knowledge base of the domain in a simple and compact way. The tool should analyze and map *DSLs* to generated code.

3.2.1 Architecture

CodeCompass is built (Figure 14) according to the server-client architecture model to be possible to supply instant reading, searching and navigation of source code in both textual and graphical formats. The server provides a Thrift¹ interface for clients through an HTTP² transport. Since the interface is specified from the Thrift interface, it is easy to add other applications or plugins to the tool (an Eclipse plugin is already implemented).

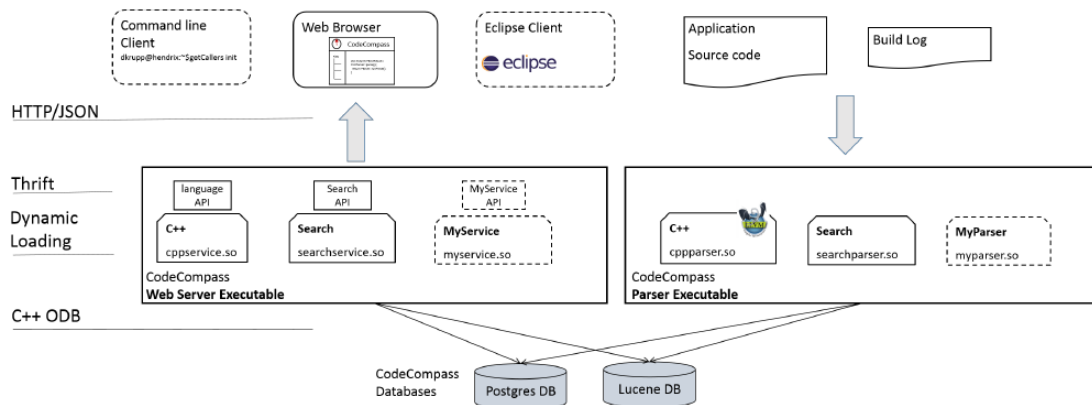


Figure 14: CodeCompass architecture

¹ Thrift is an interface definition language and binary communication protocol that is used to define and create services for numerous languages.

² HTTP is an application protocol for collaborative, distributed, hypermedia information systems.

During the parsing process, a workspace³ is physically saved as a relational database and additional files are created. This process consists of running different *parsers plugins*, each with different goals. The most important parsers are:

- **Search Parser** - It iterates recursively over all files in the corresponding project and uses Lucene⁴ to collect all the words from the source code. These words are stored in a search index, with their exact location.
- **C/C++ Parser** - Using an LLVM/Clang parser, it iterates over a database compiled in JSON⁵ that contains build actions and it stores the position and information type of AST nodes in the database. This database will be used by a C/C++ *language service* to respond to Thrift calls according to the source code.
- **Java Parser** - Using an Eclipse JDT parser, it iterates over a database compiled in JSON that contains build actions and stores the position and type of information from AST nodes in the database. This database will be used by a Java language service to respond to Thrift requests according to the source code.

Since CodeCompass has an extensible architecture it is possible to easily write new parsers in C/C++. Parser Plugins can be added to the system as shared objects. Thrift orders are served from the service plugins, on the Webserver. One or more Thrift services are implemented by a service plugin and it serves as client requests based on the information saved in the workspace. A Thrift service has a collection of methods and settings accessed from remote calls. All services have an implementation with the exception of *language service* that is implemented for C/C++, Java and Python. The *Language Service* are similar because they provide the functionality of navigating over the code base for the implemented language. The most important services are:

- **Language Service** - this service provides symbol, file, and globally query methods for the current workspace.
- **Search Service** - this service offers 4 different query types: search for *symbol definitions*⁶, search for words in text format, search for file names and suggest search for sentences based on phrase fragments.

³ A parsed snapshot of the source code is called a *workspace*.

⁴ Lucene is a free and open-source information retrieval software library, originally written in 100% pure Java.

⁵ JavaScript Object Notation is an open-standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs.

⁶ A symbol is a primitive datatype whose instances have a unique human-readable form.

3.2.2 Web User Interface

The web interface (Figure 15) is organized with a static top area, and with extendable area on the left and right side. Source code and other views can be seen in the center of the page, while other navigations are shown on the left. The workspace where the project is, the file currently open, the search area, and some generic menus to help the user, are in the upper area.

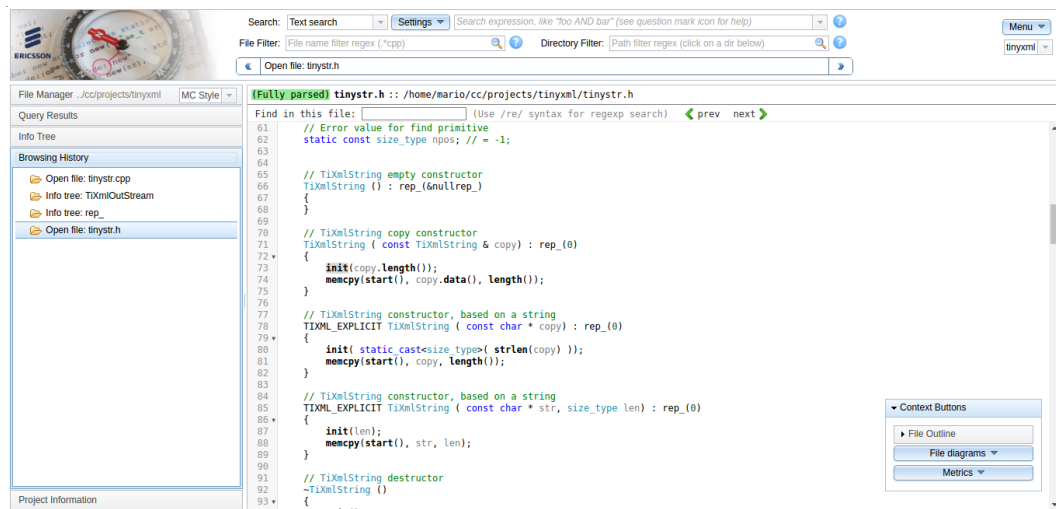


Figure 15: User Interface

3.2.3 Functionality

In this section it will be presented the features that can be used through the graphical interface.

1. **Version control visualizations** - visualization of version control information is an important support to understand software evolution. CodeCompass can also display Git commits.
2. **Code Metrics** - the tool allows you to check some metrics about the quality of the code and summarize them by directories or individual file hierarchies, of the current project.
3. **File and directory level diagrams** - it is possible to generate diagrams for directories and files, thus gaining a greater perspective of the system and its dependencies.

4. **Information about language symbols** - through the source code, the user can click on any symbol and receive information or display diagrams on it.
5. **Symbol level diagrams** - through the CodeBits (Figure 16) interactive diagram the user can browse through large call chains and type hierarchies.

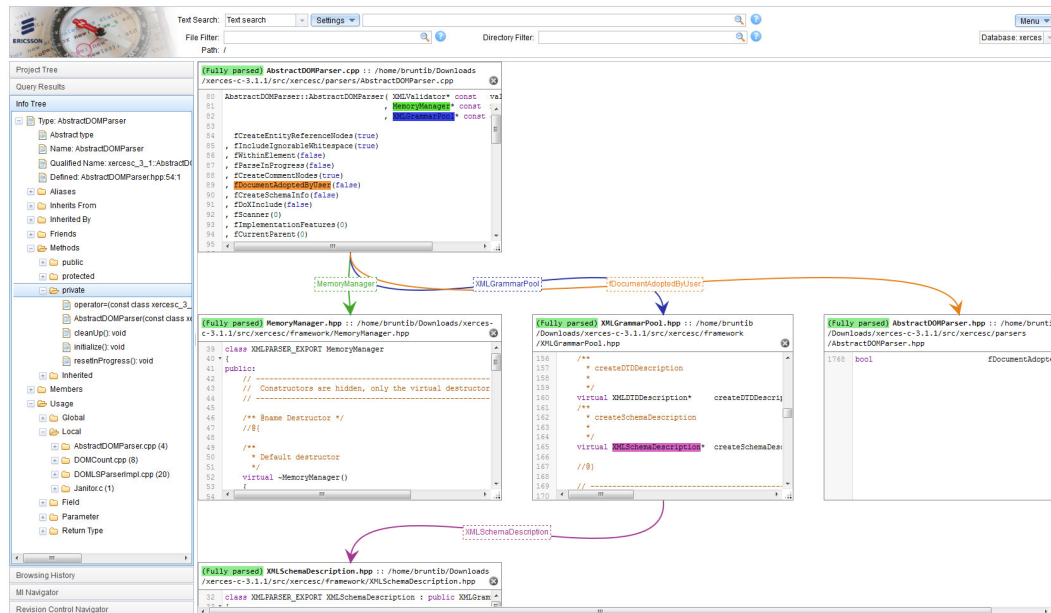


Figure 16: CodeBits example

6. **Search** - there are 4 types of research available:
 - **full text search** searches for a group of words that are followed in a block of text;
 - **definition search**, has the same syntax as the previous search but instead of text, it searches for symbol definitions;
 - **log search** looks for location in the code where the intended log messages are sent;
 - **filename search**, as the name implies, looks for the name of a particular file.
7. **Browsing history** - the user can view the browsing history over the files that he has accessed to complete the task he is performing.
8. **CodeChecker** - it allows the user to check bugs identified by Clang Tidy⁷ and Clang Static Analyzer through the CodeChecker server. With these helpers CodeCompass shows the position of the bugs and execution paths that lead to a failure.

⁷ Clang Tidy purpose is to provide an extensible framework for diagnosing and fixing typical programming errors

9. **Namespace and type catalogue** - when using Doxygen⁸, the tool saves the definitions of functions, types and variables, and provides a catalog of statements organized by the hierarchy.

3.2.4 Performance

The tool scales well in relation to the size of the analyzed code in parsing time, response times of the web server and size of the stored data.

Table 1: Performance of CodeCompass v4

	TinyXML 2.6.2	Xerces 3.1.3	CodeCompass v4	Internal Ericsson product
Source code size [MiB]	1.16	67.28	182	3 344
Search database size [MiB]	0.88	37.93	139	7 168
PostgreSQL db size [MiB]	15	190	2 144	7 729
Original build time [s]	2.73	361.77	2 024	—
Parse time [s]	21.98	517.23	6 409	—
Text/definition search [s]	0.4	0.3	0.43	2
C++ Get usage of a type [s]	1.4	2	2.3	3.1

The results of Table 1 are derived from the performance results of 4 different C/C++ applications. As one can see, parsing time is proportional to compile time.

3.2.5 User Acceptance in Real Production

Six months after the tool was released, Ericsson used it in seven projects. They observed that in projects with more than 2 million lines of code about 40% of programmers used CodeCompass at least twice a month and about 15% use it daily.

As shown in Figure 17 from ICS (2016), CodeCompass is mostly used to inspect relationships between classes, as well as to find and follow references to functions and variables.

⁸ Doxygen is a documentation generator, a tool for writing software reference documentation.

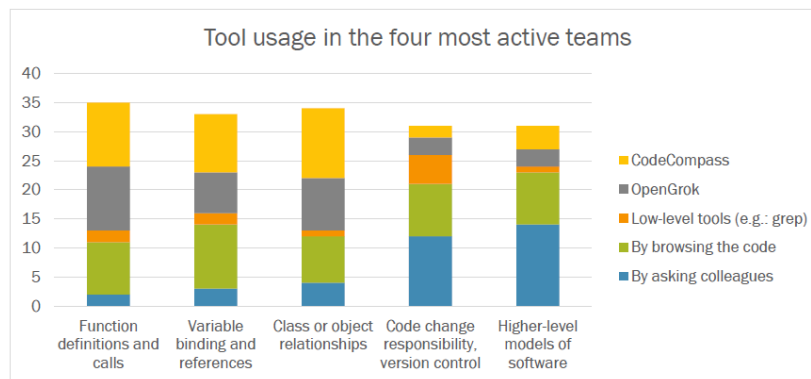


Figure 17: CodeCompass usage distribution per task

3.3 GREEN CODECOMPASS PLUG-IN

With the easy ability to extend CodeCompass I will build a plug-in to be possible to collect and visualize the energy consumed by functions and methods at runtime, of a given industrial-size project written in the two main languages (C and C++) supported by CodeCompass. In order to be possible to develop this plug-in, the project planning was divided into the following three tasks:

- **Task 1.1 - Instrument of an Application**

I will develop a tool to instrument the source code of a program, in the two main languages (C and C++) supported by CodeCompass, in order to insert librarie dependencies of RAPL and to wrap RAPL calls around the functions or methods that the user wants to profile at runtime.

- **Task 1.2 - Transform RAPL in CRAPL**

I will modify the RAPL tool in order to measure the energy consumed in the wrapped functions or methods of the instrumented program. When executing this program it will collect all energy measurements and save them in a text file.

- **Task 1.3 - Visualization data tool**

I will write a script to read the output file from the previous task and transform it in the proper format file to be evoked by a data profile visualization tool. Then it will be easy for developers to analyze the results.

3.3.1 Decisions

In order to develop a **Green CodeCompass plug-in** I needed to define the technology to use and the architecture of the program, what would be the best languages or frameworks

that I could use to reproduce the expected effect. As most **CodeCompass** and its plug-ins are implemented in **LLVM/Clang** I was advised by the members of Ericsson to learn the **LLVM/Clang** frameworks and to follow this approach. Being that Clang offers several paradigms of library and after analyzing them I ended up choosing it to implement a **LibTooling**⁹ tool. Figure 18 shows the architecture of the Green plug-in for Code Compass.

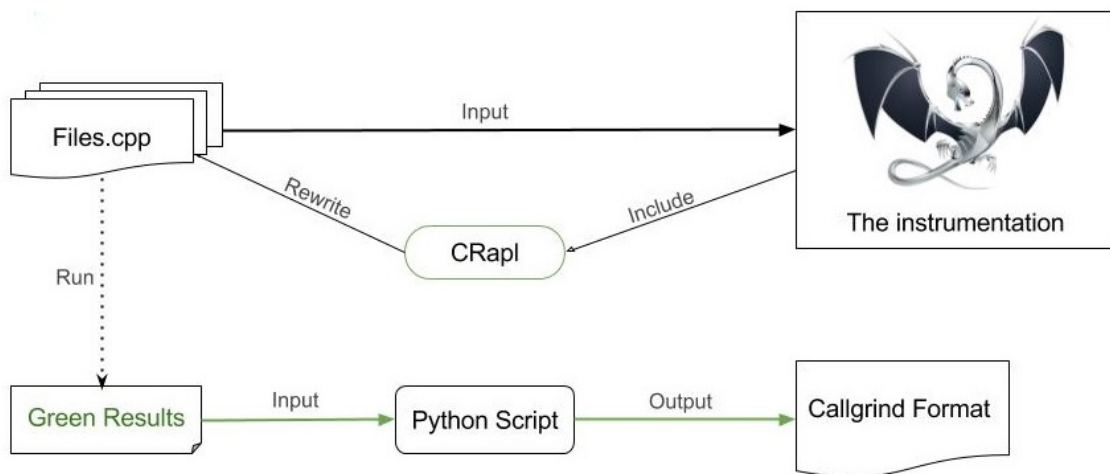


Figure 18: System architecture

First the user will send the files or the project, that he wants to analyze, to the **LibTooling** tool. It will instrument the original source code in order to measure the energy consumption of each function (inserting their information into an index text file) when the **sw** executes. After inserting the dependencies of the **CRAPL** the user needs to execute the project. After that a file with the results will be created. Then the user should send it to a script in python to put it in the necessary format to be read by the **Kcachegrind** framework.

Measurement in Different Cores

From the original RAPL code I made the necessary modifications to be possible to extract the energy values from multi-threaded programs and study a bit this topic to see if it would be possible to do a detailed study on **Kernel Threads**¹⁰.

After a brief study, I was able to draw some conclusions about energy consumption in **user level threads**. A program with a routine on data matrix 1024x1024 was executed with 1,2,4,8 and 16 threads and I found that up to 8 threads the energy consumption and the execution time decreased but it was verified that the **pp1** parameter (**GPU**) increased, this being due to the fact that it is being executed with multi-threads. In a program with

⁹ LibTooling is a library to support writing standalone tools based on Clang.

¹⁰ A kernel thread is a kernel task running only in kernel mode. It usually has not been created by `fork()` or `clone()` system calls.

smaller matrices it was found that the energy consumption increased in comparison to the sequential program.

Despite these conclusions it was found that all cores returned the same energy value and so I and the members of Ericsson, came to the conclusion that there would be no time to implement and do a detailed study on multi-threading energy consumption.

Clang Tooling

The [LLVM](#) project, started at the University of Illinois, is a collection of modular and reusable compiler and tool-chain [Lattner \(2006\)](#). [LLVM](#) has grown into an umbrella project and now includes various open source activities from compilers to static analysis. The flagship compiler for the [LLVM](#) project is Clang, the “native” compiler of [LLVM](#). Clang supports C, C++, Objective-C and Swift languages in the advanced level [Groff and Lattner \(2015\)](#). The modular, object-oriented design of Clang make it ideal for research projects require compiler-level understanding of the source code [Lattner \(2008\)](#). Having a well-defined interface for building the [AST](#), exploring it in various ways and even on-the-fly modify it, I can apply the tool-chain for instrumenting the source.

In the center of my activity is the [AST](#). The [AST](#) contains all important information (even the formatting informations via the stored positions of every element). The structure of the [AST](#) is representing the logical structure of the original program. For example the node which belongs to a `for` loop has four children: a declaration statement to introduce the loop variable, a logical expression as loop condition, an iteration expression and the body. Note that the parentheses and the semicolons in the loop header are excluded.

In the [AST](#) there are different type of nodes such as `ForStmt`, `FunctionDecl`, `BinaryOperator`, etc. These types are organised to an inheritance hierarchy which has three roots: `Decl`, `Stmt` and `Type`. Since the fundamental part of build process is compilation of translation units, the type of the root node is `TranslationUnitDecl`.

One way of using the Clang [AST](#) is to visit its nodes [Horváth and Pataki \(2015\)](#); [Clang \(2016\)](#). The visitor design pattern can be used to reach every node of the tree and perform some action when the process comes to a given type of node. Clang compiler provides a very efficient way of tree traversal by `RecursiveASTVisitor` template class. My visitor class has to inherit from this template class of which the template parameter is my class itself. The reason of this is that with this solution my class also becomes an [AST](#) visitor by the inheritance, but it does not have to pay for virtual function calls every time when running the given visitor function for the next [AST](#) node.

KCachegrind

To visualize in detail and in a user-friendly way the energy consumption values of the functions and methods tested at runtime, reported by RAPL, I had to choose a good **visualization tool** for profile data, in this case the KCachegrind¹¹ tool.

Although I did not explore the full potential of this tool, here are some features by Weidendorfer and Zenith, that weighed heavily on my choice:

- Disassembler annotation and source code views, that allows to see details of cost related to assembler instructions and source lines (left tab of figure 19);
- A tree-map view, which allows nested-call relations to be visualized, together with inclusive cost metric for fast visual detection of problematic functions (bottom right corner tab of figure 19);
- A call-graph view, which shows a section of the call graph around the selected function (top right corner tab of figure 19).

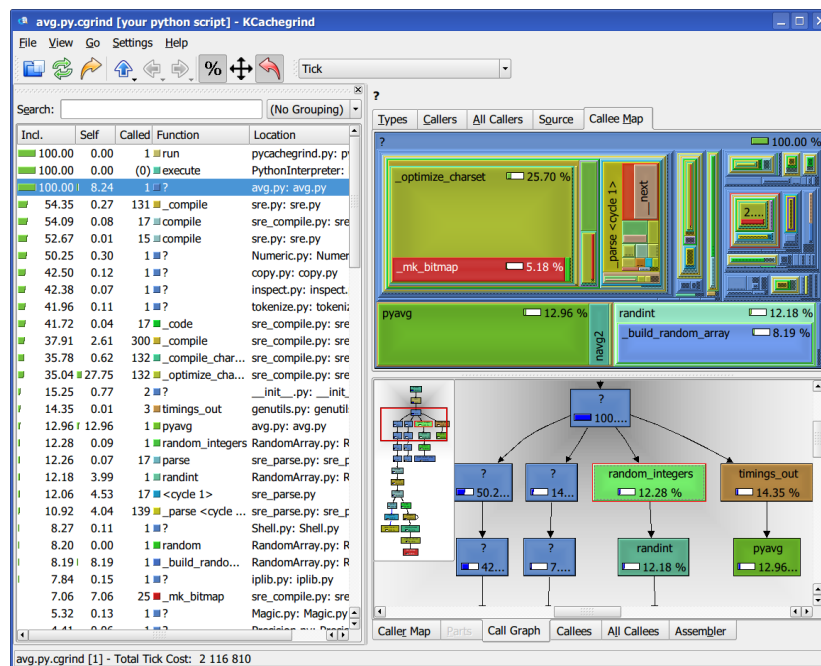


Figure 19: Visualization of Profiling Data in Kcachegrind (Weidendorfer and Zenith)

The output file needs to be converted in a **Callgrind format file** to be read by the tool, so it was necessary to create a **Python script** (detailed later in section 3.3.2) for that purpose.

¹¹ <http://kcachegrind.sourceforge.net/html/Home.html>

```

events: package pp0 pp1 dram time
#define function ID Mapping
fn=(0) usage:50
fn=(1) BaseHarnessHandlers:68
fn=(237) tassert:49
...
fl= xerces-c-3.1.4/tests/src/DOM/RangeTest/RangeTest.cpp
fn=(237)
49 15 11 0 1 1

```

As one can see above, firstly the file needs to specify what kind of values the program will read (**events**), next it is necessary to map the functions by numbers (**fn**) and then it will have all the energy measurements (in **Joules**) of the functions that were used in that execution. So, in this case, the function `tassert` that starts in line 49 of the `RangeTest.cpp` file wastes 15J in package, 11J in CPU core (pp0), 0J in CPU uncore (pp1), 1J in DRAM and it took 1 second to be executed.

3.3.2 Implementation

This section describes in detail all the implementation that was developed so that it was possible to reproduce the expected effect by the Green CodeCompass plug-in.

Firstly I will explain in detail how the instrumentation tool is implemented and what modifications it makes in the source code of a given project, next I will explain the changes that I had to make in the [RAPL](#) framework, and finally I will explain why it was also necessary to make a Python script so that the output of the plug-in is in the proper KCachegrind format.

The Instrumentation

Based on [LLVM/Clang](#), the **LibTooling** tool starts by reading the input files and will run them up my `FrontendAction`. It will create an [AST](#) with the **parsed text** of each file. For each of these generated trees it will recursively go through each node to be possible to make the necessary modifications, to **include** the CRAPL interface in the source code of the program that the user wants to analyze. An example:

```

int example(){
    int a = random();
    if(a<2)
        example();
    return a;
}

```

}

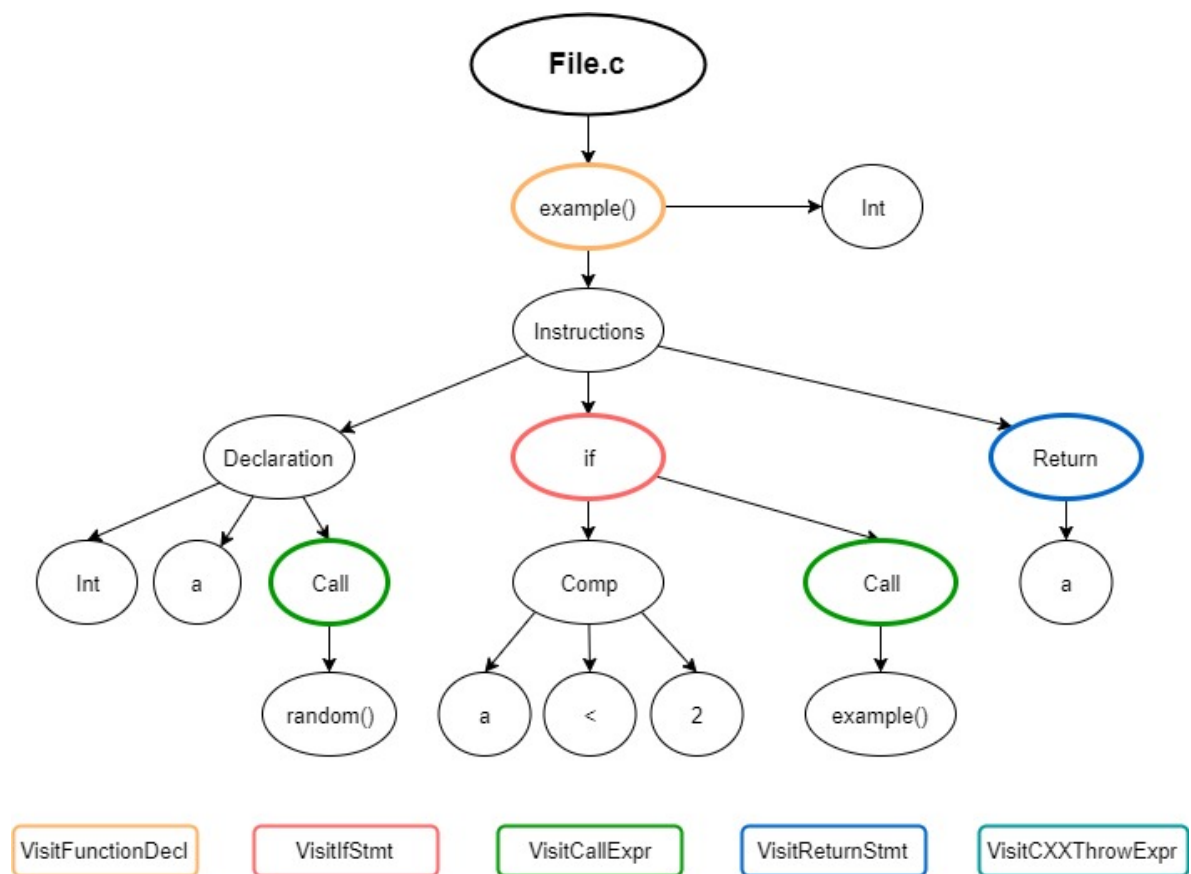


Figure 20: Example of an AST with the nodes to be read.

Using the previously source code as example to demonstrate, the instrumentation tool will undergo the necessary modifications by traversing the nodes described in the Figure 20 and detailed below:

- **VisitFunctionDecl:** It visits all the nodes that are functions. If the analyzed function does not refer to a header file and has the minimum number of statements (Instructions) that the user requested (with the optional flag -l) then the tool will insert the information (path, filename, declaration line and name) of that function into an index (array) of functions that will suffer the respective modifications until the end of the recursive reading of their child nodes. In addition it also modifies the source code with a CRAPL object initialization and inserts a `rapl_before()` call at the beginning of the functions, as one can see below with the example code:

```
int example(){
    CRapl rapl = create_rapl(0);
```

```

    rapl_before(rapl);
    int a = random();
    if(a<2)
        example();
    return a;
}

```

- **VisitIfStmt:** to maintain code consistency it is necessary to insert braces in each If or Else statement that they are not already limited by them. This because it is always required to insert a `rapl_after()` call before every single return statement of the given function. Following the code hitherto modified, the function will look like this:

```

int example(){
    CRapl rapl = create_rapl(0);
    rapl_before(rapl);
    int a = random();
    if(a<2){
        example();
    }
    return a;
}

```

- **VisitReturnStmt:** if the instrumentation tool catches a return statement anywhere in the code of the currently being analyzed function, it will insert a `rapl_after()` call to end the analysis of the energy consumption in that call. After this step the function will have the following shape:

```

int example(){
    CRapl rapl = create_rapl(0);
    rapl_before(rapl);
    int a = random();
    if(a<2){
        example();
    }
    rapl_after(0, rapl);
    return a;
}

```

- **VisitCallExpr:** the first time the totality of the Plug-in (Instrumentation + CRAPL) was tested (detailed in section 3.3.3), it was noticed that some functions consumed more energy than the main function itself, which is impossible since the main is the first to be executed and the one that finishes the program. With this output results

it was realized that I was not handling **recursive functions** so well. After a brief talk with the Ericsson's members the best solution I found for this problem was to limit blocks of code before and after the recursive call (i.e `rapl_after()` and `rapl_before()` instrumentations for each of these calls). This was one of the biggest challenges until I came up with a good generic solution, regardless of what kind of recursive call it is. After this step, the result is:

```
int example(){
    CRapl rapl = create_rapl(0);
    rapl_before(rapl);
    int a = random();
    if(a<2){
        rapl_after(0, rapl);
        example();
        rapl_before(rapl);
    }
    rapl_after(0, rapl);
    return a;
}
```

In the end of traversing each tree (of each file), the **Green CodeCompass** plug-in also inserts the dependencies of the **CRAPL libraries** and save the changes in the corresponding file (or create a new one, if the `-o = "example.cpp"` flag is enabled). So, after all the modifications the `File.c` (of Figure 20) will look like this:

```
#include <crapl/rapl_interface.h>\n"
#include <crapl/measures.h>
int example(){
    CRapl rapl = create_rapl(0);
    rapl_before(rapl);
    int a = random();
    if(a<2){
        rapl_after(0, rapl);
        example();
        rapl_before(rapl);
    }
    rapl_after(0, rapl);
    return a;
}
```

When all the instrumentation of the files is finished, it will create the **index.txt** file with the information of all the modified functions to be analyzed subsequently by CRAPL:

```
[Index:Path:Filename:Declaration_Line:Function_name]
0:xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:102:1:printFile
1:xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:138:1:error
2:xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:145:1:fatalError
3:xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:154:1:resolveEntity
4:xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:240:1:main
```

CRAPL

From the **RAPL** tool I made some modifications to be possible to read the energy consumption of several methods or functions without any conflict in their readings. For this purpose, I converted the original version of **RAPL** in C for a C++ **object-oriented version**, and then I created an interface so that **RAPL** could transform both **.c** and **.cpp** files. Thus, each instantiated **RAPL** object will read only the energy consumed by each call of that function or method.

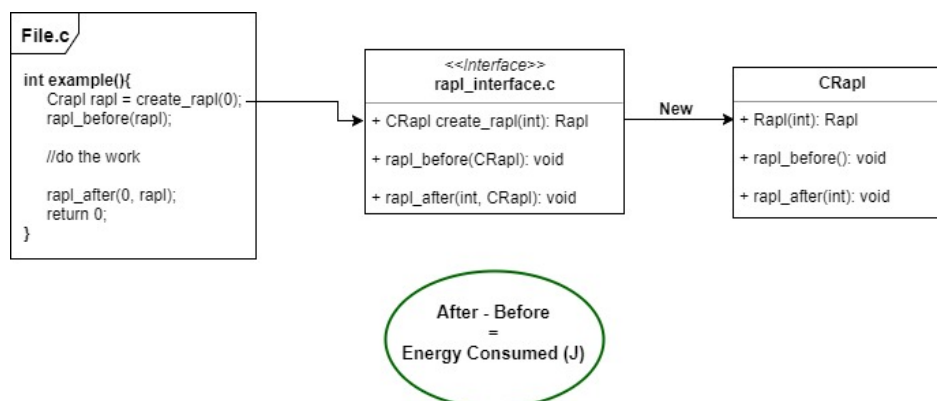


Figure 21: How CRAPL works.

For each called function or method in the execution of a program, the CRAPL interface will create a CRAPL object to read the energy consumption before the execution of the function body. After doing their job, i.e in the moment before returning or closing the function, it will read the values of energy consumption again and subtract them with the obtained values in the beginning to reach the final energy measurement of that call (Figure 21).

In addition to the required instrumentation to retrieve the values referring to the energy measurements at runtime, it is also necessary to **save** this data to be written in a file at the end of the executed program.

With the additional **measures** file (integrated in the CRAPL framework) and including the **initMeasure** and **writeMeasure** functions in the instrumentation of the main function it is possible to save the values of the energy measurements in a file.

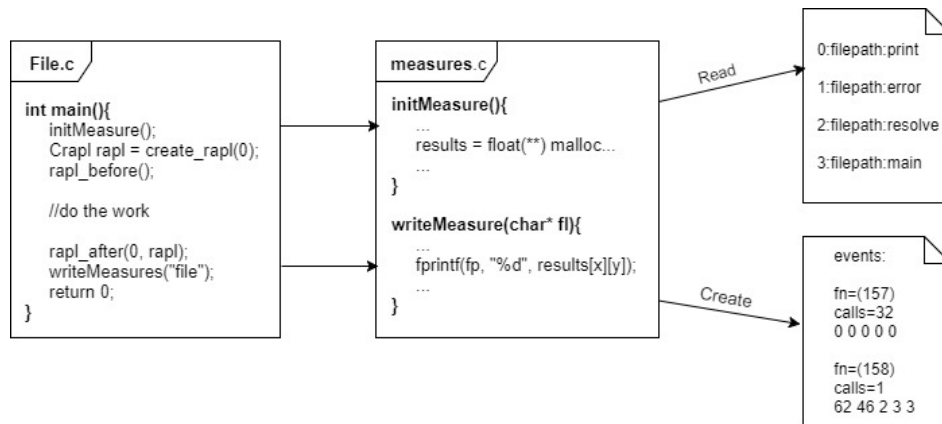


Figure 22: How energy measurement values are saved

As shown in Figure 22, in the beginning of the execution, `initMeasure` is called to **initialize** a matrix by allocating memory for N rows corresponding to the number of functions (number of lines of `index.txt` file) that were rewritten with the CRAPL methods during the previously executed instrumentation process and 5 columns referring to the values: package, `pp0`, `pp1`, `dram` and `time` (more information in section 2.3.1). At the end of the execution, the `writeMeasure` function will **write** the file with the energy consumptions referring to the functions/methods called during it.

Python Script

After running the requested project with some type of test it will be created a text file with, the name of that test and the values of the energy consumption of each function or method executed, with the following format:

```

events: package pp0 pp1 dram time
fn=(157)
calls=32
0 0 0 0 0
fn=(158)
calls=1
62 46 2 3 3 //consumed energy by events parameters
fn=(159)
calls=3

```



```

0 0 0 0 0
fn=(162)
calls=1
8390 6789 192 992 579

```

In this format, **fn** stands for the index number of the function in the index.txt file, the values of the package, ppo, pp1 and dram in **Millijoules** (10^{-3} J) mJ and the execution time in **Millisecond** (10^{-3} S) mS.

In order to read the file and visualize it in **Kcachegrind** it was necessary to create a script in python to edit this text file and save it in the correct format, the **Callgrind** format file. After executing this script, a new text file with the following format will be created:

```

events: package pp0 pp1 dram time
#define function ID Mapping
...
fn=(157) DOMTest:174
fn=(158) createDocument:184
fn=(159) createDocumentType:199
fn=(160) createEntity:211
fn=(161) createNotation:224
fn=(162) docBuilder:236
...
fl= ClangRapl/xerces-c-3.1.4/tests/src/DOM/DOMTest/DTest.cpp
fn=(157)
174 0 0 0 0 0
fl= ClangRapl/xerces-c-3.1.4/tests/src/DOM/DOMTest/DTest.cpp
fn=(158)
184 62 46 2 3 3
fl= ClangRapl/xerces-c-3.1.4/tests/src/DOM/DOMTest/DTest.cpp
fn=(159)
199 0 0 0 0 0
fl= ClangRapl/xerces-c-3.1.4/tests/src/DOM/DOMTest/DTest.cpp
fn=(162)
236 8390 6789 192 992 579

```

Since the text structure are similar, the only two differences between the old format and the new one are: the functions must be indexed and it is necessary to append the number of the declaration line of those functions, in its respective code file.

In addition to previously modifications, I have also added flag options to be possible to **normalize** the energy consumption values with **large overhead** (it will be explained in section 3.3.3). To achieve those results, it is necessary that the user also **monitor** only the **main** function and save the respective values:

```
28 12 0 7 6 //without overhead (a)
147289 114794 3090 17461 11975 //with overhead (b)
```

As one can see above, the energy consumption values of the first line **(a)** are from the monitoring of only the main function and in the second line **(b)** the values are from the main function but with the monitoring of all the other functions performed in the respective execution. With these values the script will divide each of the parameters (package, ppo, pp1 and dram) and know the **ratio** of each one to be possible to normalize the remaining functions with a **rule of three (1)**:

$$\forall n \in [0, 4], \quad \frac{a_n}{b_n} = r_n \quad (1)$$

$$\frac{a_0}{b_0} = r_0 \Leftrightarrow \frac{28}{147289} = 0.00019 \quad (2)$$

In this case 0.00019 (equation 2) will be the value of the package ratio to be multiplied by all functions (3):

$$\forall fn \in [0, 4926], \forall n \in [0, 4], \quad func[fn][n] * r[n] = normal[fn][n] \quad (3)$$

$$func[162][0] * r[0] \Leftrightarrow 8390 * 0.00019 = 1.5941mJ \quad (4)$$

For example, in function 162 (equation 4) the normalized value of the energy consumption of the package is 1.5941 mJ.

3.3.3 Plug-in Outcomes

In this section it is shown how the CRAPL final measurements results, of an example project, are visualized after the transformation of them by the python script into the correct format to be read and analyzed by the KCachegrind framework. But first, I will explain how I corrected an error when testing the implementation of the plug-in in recursive functions.

Measurement in recursive functions

After the implementation of the plug-in was complete, I tested the recursive functions in the same way as the others, but I noticed that the energy expenditures of those functions

were higher than the main function, which can not happen because the main is the first function to execute and is the last one to return. The energy spent on main is 100% percent of the energy spent in running a program.

After checking this error I had to go back to the previous step and modify the implementation (of the instrumentation tool) to be possible to wrap the recursive calls of these functions in a way to end with this **recursive mistake**. For example:

```
const char* TiXmlDocument::Parse(...){
    CRapl rapl = create_rapl(127);
    rapl_before(rapl);
    ...
    while ( p && *p ){
        TiXmlNode* node = Identify( p, encoding );
        if ( node ){
            rapl_after(127,rapl);
            p = node->Parse( p, &data, encoding );
            rapl_before(rapl);
        }
    }
    rapl_after(127,rapl);
    return p;
}
```

Although it has already a fairly large overhead due to the constant calls to the CRAPL, with this implementation we get an even **bigger overhead**. In addition, recursive functions usually spend more energy than iterative functions, because they have to push and pop the variables back into the stack when a new recursive call is made (section 2.2.4).

Energy per function

Reading the energy consumptions of all the functions called in the execution of a program is synonymous of a **large overhead** on the part of the CRAPL, i.e how many more times a function is called, the greater the percentage of total energy consumed will derive from the CRAPL calls.

As one can see from Figure 23, the KCachegrind framework tabs are:

- **Self:** energy consumption of ppo in mJ (Millijoules);
- **Function:** function name with the number of the declaration line;
- **Location:** filename of that function.

Self	Function	Location
106 936	main:90	xmltest.cpp
105 421	Parse:1043	tinyxmlparser.cpp
92 028	ReadValue:1179	tinyxmlparser.cpp
26 395	Parse:1392	tinyxmlparser.cpp
21 472	ReadText:574	tinyxmlparser.cpp
17 919	Parse:1337	tinyxmlparser.cpp
14 838	ReadName:401	tinyxmlparser.cpp
14 502	Parse:1497	tinyxmlparser.cpp
9 095	Identify:818	tinyxmlparser.cpp
7 557	Parse:704	tinyxmlparser.cpp
6 755	append:68	tinystl.cpp
6 177	~TiXmlElement:559	tinyxml.cpp
5 636	ClearThis:565	tinyxml.cpp
4 386	~TiXmlNode:147	tinyxml.cpp
4 363	Clear:169	tinyxml.cpp
3 725	LinkEndChild:186	tinyxml.cpp
3 328	Parse:1572	tinyxmlparser.cpp
1 716	LoadFile:986	tinyxml.cpp
1 409	StringEqual:534	tinyxmlparser.cpp

Figure 23: All functions measurement with overhead (ppo)

As shown in Figure 23, the values extracted from CRAPL seem nonsensical, nevertheless if I just measure the energy consumption from the main function, without the overhead, the values extracted of CRAPL for that execution are:

```
package pp0 pp1 dram time
32 13 0 8 7
```

As one can see there is a big difference between measure all functions or only the main. In this case the overhead is **8225** (the ratio of $106.396/13$), which is absurd. Because of this defect, it was decided to address more a perspective about energy consumption not by units of Joule but by **percentage** (%) in relation to the energy values of the main.

CRAPL Measurements without Overhead

To remove this large CRAPL overhead I tried to **normalize** the energy consumption (Figure 24) of all the functions using the equations mentioned above in section 3.3.2.

Self	Function	Location
13	main:90	xmltest.cpp
12	Parse:1043	tinyxmlparser.cpp
11	ReadValue:1179	tinyxmlparser.cpp
3	Parse:1392	tinyxmlparser.cpp
2	Parse:1337	tinyxmlparser.cpp
2	ReadText:574	tinyxmlparser.cpp
1	Identify:818	tinyxmlparser.cpp
1	Parse:1497	tinyxmlparser.cpp
1	ReadName:401	tinyxmlparser.cpp

Figure 24: All functions measurement without overhead (ppo)

CASE STUDIES / EXPERIMENTS

This chapter presents the hardware and software requirements to run the the Green CodeCompass plug-in. I will demonstrate step-by-step the entire flow, from its installation to the profiling of projects given as input by the user. I will also present some case studies that were taken into account during the usability analysis of the platform, as well as the discussion of these obtained results.

4.1 EXPERIMENT SETUP

In this section one can find the Intel architectures that have integrated the energy monitors chips to read the energy consumption values at different hardware levels and store it in Model-Specific Registers to be accessed by the **RAPL** interfaces. Then, it will be presented how to install and run the Green CodeCompass plug-in.

First of all, one can examine the Green CodeCompass Plug-in source code and download it in the following link: <https://github.com/Galay125/energy-analysis>

4.1.1 *Hardware Prerequisites*

As previously stated (in the section 2.3.1), **RAPL** is only supported by recent **Intel architectures** such as i5 and i7 CPUs. One can verify it in the Figure 25. Not all processors have the ability to make available all the fields accessed by **RAPL** interfaces about the energy consumption of some pieces of hardware.

The execution tests were all run on a laptop (Asus X555LJ) with the following characteristics:

- 3.16.0-38-generic GNU/Linux (LinuxMint 17.2)
- Intel® Core™ i7-5500U Processor 2.40GHz Dual Core
- Memory 8 GB (4 GB DDR3 Onboard + 4GB DDR3)

Name	Family	Model	package	PP0 (usually cores)	PP1 (usually GPU)	DRAM	PSys	powercap	perf_event	PAPI
Sandybridge	6	42	Y	Y	Y	N	N	3.13 (2d281d8196)	3.14 (4788e5b4b23)	yes
Sandy Bridge EP	6	45	Y	Y	N	Y	N	3.13 (2d281d8196)	3.14 (4788e5b4b23)	yes
Ivy Bridge	6	58	Y	Y	Y	N	N	3.13 (2d281d8196)	3.14 (4788e5b4b23)	yes
Ivy Bridge EP ("Ivy Town")	6	62	Y	Y	N	Y	N	no	3.14 (4788e5b4b23)	yes
Haswell	6	60	Y	Y	Y	Y	N	3.16 (a97ac35b5d9)	3.14 (4788e5b4b23)	yes
Haswell ULT	6	69	Y	Y	Y	Y	N	3.13 (2d281d8196)	3.14 (7fd565e27547)	yes
Haswell	6	70	Y	Y	Y	Y	N	4.6 (462d8083f)	4.6 (e1089602a3bf)	yes
Haswell EP	6	63	Y	?	N	Y	N	3.17 (64c7569c065)	4.1 (64552396010)	yes
Broadwell	6	61	Y	Y	Y	Y	N	3.16 (a97ac35b5d9)	4.1 (44b11fee517)	?
Broadwell-H	6	71	Y	Y	Y	Y	N	4.3 (4e0bec9e83)	4.6 (7b0fd569303)	?
Broadwell-DE	6	86	Y	?	?	Y	N	3.19 (d72be771c5d)	4.7 (31b84310c79)	?
Broadwell EP	6	79	Y	?	?	Y	N	4.1 (34dfa36c04c)	4.6 (7b0fd569303)	?
Skylake	6	78	Y	?	?	Y	?	4.1 (5fa0fa4b01)	4.7 (dcee75b3b7f02)	?
Skylake H S	6	94	Y	?	?	Y	?	4.3 (2cac1f70)	4.7 (dcee75b3b7f02)	?
Skylake Server	6	85	Y	?	?	Y	?	4.8???	4.8 (348c5ac6c7dc11)	no
Kabylake	6	142,158	Y	?	?	Y	?	4.7 (6c51cc0203)	4.11 (f2029b1e47)	no
Knights Landing	6	87	Y	?	?	Y	N	4.2 (6f066d4d2)	4.6 (4d120c535d6)	?
Knights Mill	6	133	Y	?	?	Y	N	?	4.9 (36c4b6c14d20)	?
Atom Goldmont (Apollo Lake?) "Broxton"	6	92	Y	Y	Y	Y	N	4.4 (89e7b2553a)	4.9 (2668c6195685)	no
Atom Braswell	6	76	?	?	?	?	N	3.19 (74af752e4895)	no	no
Atom Tangier	6	74	?	?	?	?	N	3.19 (74af752e4895)	no	no
Atom Annidale	6	90	?	?	?	?	N	3.19 (74af752e4895)	no	no
Atom Valleyview	6	55	?	?	?	?	N	3.13 (ed93b71492d)	no	no

Figure 25: x86 Intel Architectures that have RAPL interface(Weaver, 2015).

Since the initial architecture is **Broadwell** (model 61) I was able to test all the possible parameters provided by **RAPL**, namely: package, pp0, pp1 and dram.

4.1.2 Software Prerequisites and Configuration

To run Green CodeCompass plug-in you need **LLVM** + Clang, the installation can be found here:

http://clang.llvm.org/get_started.html

Configuration and building

The configuration of the plug-in is done simply using **CMake**. Make sure that the **LLVM** binaries are in your PATH: llvm-config is invoked during the configuration. Use **GCC 4.8** or later to compile.

```
mkdir build
cd build
cmake ..
make or sudo make install
```

Usage

Running the entire platform involves 3 steps:

- **Instrumentation**

To instrument a project in its own directory ("./") the following script is provided:

```
instru -l=5 -d="." .cpp --
```

CRapl_Gen directory is created with file index.txt !

One can also use the follow flags:

```
.cpp - extension.
[optional] -l = number of minimum statements;
[optional] -o = output file;
[optional] -d = directory;
[optional] -fns = if the user just want to measure one function,
type the name of it.
```

- **CRAPL**

The following Linux commands install the CRAPL libraries in the system:

```
cd Rapl\crapl
sudo make install
```

Edit **makefile** to get the libraries dependencies, for example:

```
RAPL:= /home/username/Documents/Rapl
INCS := -I$(RAPL)
RAPLSRCS := ${RAPL}/crapl/measures.o ${RAPL}/crapl/rapl_interface.o
${RAPL}/crapl/rapl.o
SRCS := ${RAPLSRCS}
LDADD = ... ${RAPLSRCS}
DEFAULT_INCLUDES = ... -I$(RAPL)
```

- **Execute your project:**

```
make install
sudo modprobe msr // load MSR driver
sudo <exec_file>
cd CRapl_Gen and check your output (<exec_file>.txt)
```

Python Script

At this point, one need to convert the results extracted by CRAPL, when the instrumented C++ program is executed, to the **Callgrind Profile Format**¹ file to visualize it in the **KCachegrind** framework:

```
python toCallgrind.py -i ../tinyxml/CRapl_Gen/index.txt -f
../tinyxml/CRapl_Gen/xmltest.txt -d tinyxml -o cache.txt
```

¹ <http://kcachegrind.sourceforge.net/html/CallgrindFormat.html>

One can also use the follow flags:

```
-i = index of the functions;
-f = output file of the results;
[optional] -d = directory;
[optional] -o = output with callgrind format;
[optional] -m = if the user want to normalize the results you can give a file
with the values of energy consumption just in the main
```

KCachegrind visualization

First, one need to install the KCachegrind framework to **visualize** the output from the python script in the correct format file. The installation can be found in the following link: <http://kcachegrind.sourceforge.net/html/Download.html>

Then run the tool with the specific file as input:

```
Kcachegrind name_of_file.txt
```

4.2 RESULTS

This section presents an analyze on the obtained results when testing the Green CodeCompass plug-in on some projects requested by the Ericsson coordinator.

Currently our tool has been tested and analyzed intensively on two projects, **TinyXml**² (small size) and some **Xerces-c-3.1.4**³ (medium size) samples/tests, both XML parsers written in C++.

4.2.1 *TinyXml*

TinyXml is a small-sized project and just have one directory, so it is not necessary to use the optional directory flag (-d) to traverse nested directories when running the **instrumentation** tool on all (*) of the .cpp (c++) program files, so one just need to execute the follow command:

```
instru *.cpp --
```

After this step it is necessary to modify the project **makefile**, adding the dependencies of the CRAPL libraries:

² <http://www.grinninglizard.com/tinyxml/>

³ <https://xerces.apache.org/xerces-c/>


```

RAPL:= /home/user/Documentos/Rapl
INCS := -I$(RAPL)
RAPLSRCS := ${RAPL}/crapl/measures.o ${RAPL}/crapl/rapl_interface.o
${RAPL}/crapl/rapl.o
SRCS := tinyxml.cpp tinyxmlparser.cpp xmltest.cpp tinyxmlerror.cpp
tinystr.cpp ${RAPLSRCS}

```

Then you need to **compile** the whole project:

```
make install
```

And execute the only test that TinyXml offers, with administrator privileges using sudo to load **MSR** driver and to get the energy consumption values at runtime:

```

sudo modprobe msr
sudo ./xmltest

```

During the previously execution it was created a text file referring to the energy values of each called function but it is still necessary to **convert** (with the Python Script) it in the correct format file to be visualized in the KCachegrind framework:

```

python toCallgrind.py -i ../tinyxml/CRapl_Gen/index.txt -f
../tinyxml/CRapl_Gen/xmltest.txt -d tinyxml -o kcache.txt

```

In the end, it is just required to execute the following command to allow the user to **visualize** (for example Figure 27) the results of this green profiling in KCachegrind:

```
Kcachegrind kcache.txt
```

Self	Function	Location
28	main:90	xmltest.cpp
27	Parse:1043	tinyxmlparser.cpp
23	ReadValue:1179	tinyxmlparser.cpp
6	Parse:1392	tinyxmlparser.cpp
5	ReadText:574	tinyxmlparser.cpp
4	Parse:1337	tinyxmlparser.cpp
3	Parse:1497	tinyxmlparser.cpp
3	ReadName:401	tinyxmlparser.cpp
2	Identify:818	tinyxmlparser.cpp
1	Clear:169	tinyxml.cpp
1	ClearThis:565	tinyxml.cpp
1	Parse:704	tinyxmlparser.cpp
1	append:68	tinystr.cpp
1	~TXmlElement:559	tinyxml.cpp
1	~TXmlNode:147	tinyxml.cpp
0	Accept:1156	tinyxml.cpp
0	Accept:1314	tinyxml.cpp
0	Accept:1360	tinyxml.cpp

Figure 26: Tinyxml results without overhead

Self	Function	Location
147 289	main:90	xmltest.cpp
144 855	Parse:1043	tinyxmlparser.cpp
126 089	ReadValue:1179	tinyxmlparser.cpp
35 406	Parse:1392	tinyxmlparser.cpp
28 660	ReadText:574	tinyxmlparser.cpp
25 824	Parse:1337	tinyxmlparser.cpp
20 460	ReadName:401	tinyxmlparser.cpp
19 371	Parse:1497	tinyxmlparser.cpp
12 554	Identify:818	tinyxmlparser.cpp
10 289	Parse:704	tinyxmlparser.cpp
8 613	append:68	tinystr.cpp
8 035	~TXmlElement:559	tinyxml.cpp
7 351	ClearThis:565	tinyxml.cpp
6 117	~TXmlNode:147	tinyxml.cpp
5 843	Clear:169	tinyxml.cpp
5 062	LinkEndChild:186	tinyxml.cpp
4 820	Parse:1572	tinyxmlparser.cpp
2 363	StringEqual:534	tinyxmlparser.cpp

Figure 27: Tinyxml results with overhead

Due to a high number of calls from all functions (total of 181.889), I received a large overhead from CRAPL. In order to remove this overhead (as mentioned above in section 3.3.2) first I would have to test only the main function to collect the correct values of the energy

consumed by the program and then I have to run the python script to do the necessary normalizations (rule of three) of those energy results.

In this phase I will mainly present the results with the overhead, and since the KCachegrind reads only integers, the units will be presented in **Milijoules** (10^{-3} J) mJ.

As shown in Figures (26 and 27) when I call the CRAPL in all functions of TinyXml we get a large overhead (the number of calls to RAPL interfaces is too high) but if the main goal of the plug-in is to realize which are the functions that spend more energy, this process is necessary. Regardless, overhead does not prevent us from doing an analysis on the TinyXml project (by percentage of energy consumed per function).

I can verify that the **Parser** function (at line 1043 of tinyxmlparser.cpp) and its child nodes (nested functions calls) represent about 97% of the energy (Package) consumed during the execution of the xmltest. The **Parser** function uses recursion, which means that at the beginning of each function the stack have to push and pop the arguments (as mentioned in section 2.2.4). This process causes the function to have a longer execution time and consequently a higher energy consumption, according to studies made by Fakhar et al. (2012).

4.2.2 Xerces-c-3.1.4

Xerces has a complex structure and it is bigger than the previous project. To be possible to go through and reach all nested directories (specifically *src*, *tests* and *samples*) that contains .c or .cpp files to be instrumented, it was necessary to use the **optional flag** for directories (-d=".") in the command to run the instrumentation tool.

After the executed steps explained in the previous sections (4.1.2 and 4.2.1) and the project already compiled, the user is able to run 16 samples⁴ and some tests.

Tests	Functions					
	Main	Initialize:162	BuildR:93	getUni:229	Terminate:328	match:995
CreateDOM	296,2	292	194,1	5,6	3,6	1,5
DOMCount	364,1	352,8	225,7	6,7	2,8	1,9
DOMPrint	318,3	303,9	198,5	6,6	3,3	1,2
DTest	973	329,5	210,1	6,6	3,8	399,2
RangeTest	286	275,8	179,7	5,5	3,2	0,75
Traversal	335,9	331,1	215,6	6,8	3,5	0,8
Total (%)	100%	75%	48%	1,5%	0,8 %	16%

Table 2: Energy Consumption of functions executed in some tests

⁴ <https://xerces.apache.org/xerces-c/samples-3.html>

So, with about **4926** instrumented functions it was performed 6 tests of the Xerces project and these were the energy (Package) results (Table 2) obtained for some executed functions without removing the overhead (in **Joules**).

From these results (Table 2) I can not draw great conclusions except that most of the functions with high energy consumption are child nodes of the **Initializer:162**, because it is a function with only one call and with the highest percentage of energy spent. Also I can verify that the best and most interesting case to be analyzed in detail is the **DTest** because of the irregularity values of the function **match:995**.

Table 3 shows some results (Package) about the most expensive functions in DTest.

Functions	PKG (J)	%	Calls	Energy/Calls (J)	Time (S)
Main:840	973	100%	1	973	90
testRegex:5393	569,3	58%	1	569,3	55
matches:517	542,9	56%	84	6,5	52
match:995	399,2	41%	52448	0,008	39
Initialize:162	329,5	34%	1	329,5	30

Table 3: Detailed energy consumption from DTest

As one can see in Table 3, the function **match:995** has a very high energy consumption compared to the other tests due to be used many times (52.448 calls). So, from this table it can be said that the **match:995** function spends less energy in relation to the rest (only 0.008J per call). Knowing that this function is used mostly by the function **matches:517** I can conclude that 70-75% of the energy consumed by **matches:517** derives from **match:995**.

4.3 DISCUSSION

In this section it will be discussed the validity of the Green CodeCompass plug-in results obtained in the performance of **TinyXml** project. How with some techniques and, the implementation of new flags and options in the tool it is possible to study their own validity.

4.3.1 Validating the Measurements

As previously mentioned, the overhead of the CRAPL readings is one of the main problems for the validation of the energy values in units by the plug-in, not being able to ignore this problem I decided to make measurements of the energy consumption only of the main function and the functions that the tool claims to consume more energy (one by one). In order to be possible this procedure it was necessary to modify the instrumentation tool and add a optional flag **-fns** to be possible for the user to instrument only the function that he wants to study.

After applying this method, I re-run xmltest 5 times, for each function represented in the Table 4.

Functions	Calls	Tests Avg	Main Avg	Tests	Global Avg	Global	Error
Parse:1043	1007	732	769	95%	144 855	98%	3%
ReadValue:1179	865	416,4	482,6	86%	126 089	86%	0%
Parse:1392	1499	92,8	859	11%	35 406	24%	13%
ReadText:574	1988	147,2	1404,2	10%	28 660	19%	9%
Parse:1337	482	30,8	296,6	12%	25 824	17%	5%

Table 4: Validation of results in TinyXml (values represented in mJ)

As can be seen, in Table 4 the columns can be defined as:

- **Tests Avg:** the average values of each measured function in the executed xmltest;
- **Main Avg:** the average values of the main function when measured only with the respective function (of that line);
- **Tests:** the percentage of the average value in relation to the total (value of the main), for example 732mJ is 95% of 769mJ.
- **Global Avg:** the measured values from Figure 27, in the case of study presented in section 4.2.1;
- **Global:** the percentage of the previously values in relation to the total (main) of the Figure 27;
- **Error:** the absolute value of the difference between the Global percentage and Tests percentage.

Following the values of Table 4, the overhead of doing the measurement process of all the functions is enormous in relation of measuring one by one, however, the overhead continues to exist but I can now obtain better conclusions regarding the percentage that each function exerts on the total energy consumption in the execution of xmltest.

From the **Error** column I can conclude that although the overhead exists the percentages do not vary much (between 0-13% in this case).

According to this study I can say that the **rule of three** (Equation 1, 2, 3, 4) applied in the energy values of project TinyXml to normalize them (to remove overhead of CRAPL calls) are valid in the tested case (xmltest). So I can conclude that the energy measurements from Figure 26 ("self" tab) are the correct energy consumption values (in mJ) for those functions.

In addition, readings of the RAPL interface are **hardware-based**. It is impossible to isolate the energy consumption due to running the **operating system**, or running **applications**.

CONCLUSION

5.1 CONCLUSIONS

Our tool can be a good complement for C/C++ programmers who are interested in understanding the energy consumption of their programs. This is a theme that may not emerge much when small programs are used by a single machine but can have a positive environmental impact, or even reduce economically energy costs, when we talk about large scale projects used in servers or millions of personal computers around the world.

At the end of Green CodeCompass plug-in development and validation, I was able to answer the three research questions, presented in section 1.2:

- RQ1: Can we instrument a all C or C++ software system to add the [RAPL](#) interface without compromise the execution of the programme?

The implemented instrumentation (section 3.3.2) has no conflict over the work performed of the functions or methods belonging to a project or file, instrumented. On the other hand, the performance of the functions will not be the same, when you add RAPL calls you are increasing both the execution time and its energy consumption. Either way, the only goal of adding RAPL calls to a system is just to study its energy consumption.

- RQ2: Can we measure the energy of all the functions/methods of a project to easily check which ones are wasting more energy?

With the CRAPL (object-oriented version) framework (section 3.3.2), it is possible to measure the energy consumption of various functions during the execution of a system, without any collision between RAPL calls. From the obtained results of those measurements and after being converted to the appropriate file format (section 3.3.2), you can visualize the functions that spent more energy during the execution of the respective program, in the KCachegrind tool (section 3.3.3). When verifying a certain

inconsistency of the energy values referring to recursive functions (section 3.3.3), it was necessary to modify the instrumentation algorithm to repair this error.

- RQ3: Can such techniques be implemented as a plugin of the CodeCompass tool? How efficient and effective are such techniques when handling industrial-size `sw` applications?

By adding the optional flag (-d) to navigate nested directories during instrumentation, it was possible to study more complex projects (section 4.2.2). The larger a system, the bigger the overhead of the CRAPL calls during its execution. So, the obtained results are not conclusive at the units (Joules or Millijoules) level (unless you only measure the power consumption of the main function) but following a prespective (validated in section 4.3.1) on the rate of use of the functions (in percentage) during the execution, it was possible to draw some conclusions regarding the higher energy consumption by recursive functions (section 4.2.1). Although the Green CodeCompass plug-in is fully operational there was no time, during my stay in Budapest, to integrate it into the CodeCompass environment but according to its creators (members of Ericsson Budapest) it would be an easy task since CodeCompass has an extensible architecture (section 3.2.1).

Finally, I can say that the Erasmus program was a really important academic experience for me. It was possible not only to implement this project but was also very important to my own personal and professional growth. In the end, as a result of the partnership between Ericsson, the University of Minho and the Faculty of Informatics of Eötvös Loránd University, it was possible to submit an article at the [SQAMIA 2017](#) conference. It was held between 11 and 13 September in Belgrade, Serbia and the article was presented by Dr. Zoltán Porkoláb, with a positive feedback from the participants.

5.2 PROSPECT FOR FUTURE WORK

This project can serve as a good basis for the construction and development of various perspectives of energy code analysis. Some ideas that may be applied in the future:

- **Collect the Control Flow** - it would be interesting to shape CRAPL so that it would pick up the paths of the executed functions and so, we could study the results in more detail through the visualization of more illustrative graphs in Kcachegrind.
- **Collect energy from standard functions** - collecting energy measurements from standard libraries functions would also give us a good perspective on which functions or methods or structures we should choose depending on how much data our implementations will receive.

- **Collect only the first call of a recursive function** - although I have already implemented this version, there was no time to make a study and draw conclusions about this approach. By theory, it would be a good prospect in reducing the overhead of CRAPL calls in recursive functions.
- **Intensive study on the overhead of CRAPL calls** - even knowing that the results in percentage are good approximations of reality, it would be more meaningful to show the results in the appropriate units and without the overhead that is gained progressively in each call of the CRAPL while testing programs of industrial size. Therefore, it would be a good study to realize how much energy is consumed by each CRAPL call and in the end to withdraw this value to the total energy consumption of each function/method executed.
- **More options in the Python Script** - instead of just creating an output in the callgrind format to be read in KCachegrind, we could have more ways to visualize the obtained results by implementing more output options in the python script, such as creating .csv files to be read in excel.
- **Threads to measure while running** - one of the perspectives and ideas of Professor João Saraiva to reduce or eliminate the overhead at the level of the CRAPL calls, would be to implement the CRAPL to make the measurements in parallel with the execution of the program to be tested. Briefly, the idea is to create a thread that measures energy consumption every second.
- **Intensive study in Parallel Programming** - although some Ericsson members have already write an article on energy consumption in Multicore processors, it would be a good continuation to do a study about industrial-sized projects and to see if a parallel approach would be more advantageous in relation to a sequential approach, at the level of methods and functions.
- **Extend the plug-in for IDEs** - it might be more practical to trying to extend this plug-in so that it is easily used while programmers are developing in their integrated development environment. They could then test their functions and methods while developing their projects.

BIBLIOGRAPHY

- Jrapl - a framework for profiling energy consumption of java programs. <https://github.com/kliu20/jRAPL>.
- Codecompass, June 2016. URL <https://github.com/Ericsson/CodeCompass>.
- Codecompass: An open software comprehension framework. 2016.
- Sérgio Daniel Tristão Alves et al. Green computing. 2012.
- Ayse Basar Bener, Maurizio Morisio, and Andriy Miranskyy. Green software. *Ieee Software*, 31(3):36–39, 2014.
- Coral Calero and Mario Piattini. *Green in Software Engineering*. Springer, 2015.
- Clang. Doxygen documentation of recursive ast visitors. 2016. URL http://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html.
- Marco Couto, Tiago Carção, Jácome Cunha, JoãoPaulo Fernandes, and João Saraiva. Detecting anomalous energy consumption in android applications. In FernandoMagno Quintão Pereira, editor, *Programming Languages*, volume 8771 of *Lecture Notes in Computer Science*, pages 77–91. Springer International Publishing, 2014. ISBN 978-3-319-11862-8.
- Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. Towards a green ranking for programming languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP 2017*, pages 7:1–7:8, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5389-2. doi: 10.1145/3125374.3125382. URL <http://doi.acm.org/10.1145/3125374.3125382>.
- Martin Dimitrov, Carl Strickland, Seung-Woo Kim, Karthik Kumar, and Kshitij Doshi. Intel® power governor. <https://software.intel.com/en-us/articles/intel-power-governor>, 2015. Accessed: 2015-10-12.
- STAR ENERGY. Energy star®. *History: ENERGY STAR*, 2011.
- Ericsson. Ericsson history, nov 2016. URL <https://www.ericsson.com/en/about-us/history>.
- Faiza Fakhar, Barkha Javed, Raihan ur Rasool, Owais Malik, and Khurram Zulfiqar. Software level green computing for large scale systems. *Journal of Cloud Computing: Advances*,

- Systems and Applications*, 1(1):4, 2012. ISSN 2192-113X. doi: 10.1186/2192-113X-1-4. URL <http://dx.doi.org/10.1186/2192-113X-1-4>.
- J. Groff and C. Lattner. Swift’s high-level ir: A case study of complementing llvm ir with language-specific optimization. Lecture at The ninth meeting of LLVM Developers and Users, 2015.
- Miguel Guimarães, João Saraiva, and Orlando Belo. Categorização do consumo de energia em sistemas de povoamento de data warehouses. In *Atas da Conferência da Associação Portuguesa de Sistemas de Informação*, volume 15, pages 460–474, 2016.
- Robert R Harmon and Nora Auseklis. Sustainable it services: Assessing the impact of green computing practices. In *PICMET’09-2009 Portland International Conference on Management of Engineering & Technology*, pages 1707–1717. IEEE, 2009.
- Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Seep: exploiting symbolic execution for energy-aware programming. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, page 4. ACM, 2011.
- G. Horváth and N. Pataki. Clang matchers for verified usage of the C++ Standard Template Library. *Annales Mathematicae et Informaticae*, 44:99–109, 2015. URL http://ami.ektf.hu/uploads/papers/finalpdf/AMI_44_from99to109.pdf.
- Intel Intel. and ia-32 architectures software developer’s manual, 2011. *Intel order Number*, 64, 64.
- C. Lattner. Llvm and clang: Next generation compiler technology. Lecture at BSD Conference 2008, 2008.
- Chris Lattner. Introduction to the llvm compiler infrastructure. In *Itanium Conference and Expo*, 2006.
- LuÃs Gabriel Lima, Gilberto Melfe, Francisco Soares-Neto, Paulo Lieuthier, JoÃo Paulo Fernandes, and Fernando Castor. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER’2016)*, pages 517–528. IEEE, 2016. ISBN 978-1-5090-1855-0.
- Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering*, pages 316–331. Springer, 2015.
- Sara S Mahmoud and Imtiaz Ahmad. A green model for sustainable software engineering. *International Journal of Software Engineering and Its Applications*, 7(4):55–74, 2013.

- C Mines. Reasons why cloud computing is also a green solution". *GreenBiz, Web: <http://www.greenbiz.com/blog/2011/07/27/4-reasons-why-cloud-computing-also-greensolution>*.
- Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software, GREENS '16*, pages 15–21, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4161-5. doi: 10.1145/2896967.2896968. URL <http://doi.acm.org/10.1145/2896967.2896968>.
- Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João P. Fernandes, and João Saraiva. Mind the leak: Helping programmers improve the energy efficiency of source code (short paper). In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*, Buenos Aires, Argentina, 2017a. ACM and IEEE CS.
- Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017*, pages 256–267, New York, NY, USA, 2017b. ACM. ISBN 978-1-4503-5525-4. doi: 10.1145/3136014.3136031. URL <http://doi.acm.org/10.1145/3136014.3136031>.
- Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31. ACM, 2014.
- Inc. Qualcomm Technologies. Trepn power profiler.
- J Scheild. A history of green computing, its use, the necessity and the future. *Available: tp.igit.etenkirtit.tetki.tatigt.atticles.42s*, 2011.
- Energy Star. Put your computers to sleep. URL https://www.energystar.gov/products/low_carbon_it_campaign/put_your_computers_sleep.
- Amber Statham, James Elkins, and Sidney Blaney. Green computing hardware, 2012. URL <https://prezi.com/bahdong--r7o/green-computing-hardware/>.
- Efrain Turban, Dave King, J Lee, and Dennis Viehland. Chapter 19: Building e-commerce applications and infrastructure. *Electronic Commerce A Managerial Perspective*, page 27, 2008.
- Vince Weaver. Linux support for power measurement interfaces, 2015. URL http://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html.
- Josef Weidendorfer and F Zenith. The kcache grind handbook.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.