

Marco Rafael Linhares Couto. **Supporting Software Developers in Making Energy Saving Decisions**. Uminho | 2020

Universidade do Minho
Escola de Engenharia

Marco Rafael Linhares Couto

Supporting Software Developers in Making Energy Saving Decisions

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto

Universidade do Minho

universidade de aveiro

U.PORTO

Outubro de 2020



Universidade do Minho
Escola de Engenharia

Marco Rafael Linhares Couto

Supporting Software Developers in Making Energy Saving Decisions

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto



Universidade do Minho





Trabalho efetuado sob a orientação do
Professor Doutor João Alexandre Saraiva
e do
Professor Doutor João Paulo Fernandes

Outubro de 2020

Acknowledgements

I wish to express my dearest gratitude to everyone who accompanied me in this journey. A big thank you to my outstanding supervisors, Prof. João Saraiva and Prof. João P. Fernandes, and my friends and colleagues Prof. Jácome Cunha and Rui Pereira. I also want to thank Prof. Cristina V. Lopes, Prof. Simão Melo de Sousa, Prof. Paulo Borba and Luís Cruz for their valuable help.

To my family, thank you for all the fun moments, but also for your guidance. And a very heartfelt thank you to my amazing parents, for all the hard work you put into raising me. It ultimately lead me here, and I hope it makes you guys proud. A special thanks to my brother and sisters, and all my cousins, for your friendship and encouragement, and for making me proud of you every day. And at last, by certainly not least, a huge thank you to the amazing mother of my child, for bearing up with me, specially in the hard times, and for helping me build a happy and stable life environment. I am so grateful for having all of you in my life!

Finally, this work was partially funded by FCT - Foundation for Science and Technology, the Portuguese Ministry of Science, Technology and Higher Education, through national funds, and co-financed by the European Social Fund (ESF) through the Operational Programme for Human Capital (POCH), with scholarship reference SFRH/BD/132485/2017. Additionally, funding was also provided by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», by the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, from projects «UID/EEA/50014/2013» and Green Software Lab (ref. «POCI-01-0145-FEDER-016718»), and by the Luso-American Foundation in collaboration with the National Science Foundation with grant FLAD/NSF ref. 300/2015 and ref. 275/2016.





















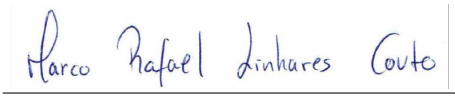
STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, October 19, 2020

Full Name: Marco Rafael Linhares Couto

Signature: 

Abstract

Supporting Software Developers in Making Energy Saving Decisions

In the last decade, energy consumption analysis and improvement in software has been establishing as a new key concern for developers. Nevertheless, as recent studies demonstrate, developers are still struggling to understand how to analyze and improve the energy efficiency of their code. Aware of this, the research community has been continuously working on providing new tools, techniques, and findings that developers can rely on.

With this thesis, our main target is to further extend the knowledge and mechanisms available for developers to statically reason about the energy efficiency of their code. Our contributions can be grouped in two topics: energy behavior prediction and energy-aware software evolution.

To tackle energy prediction, we developed a new approach which combines static program analysis/verification with energy models, with the goal of providing accurate energy estimations for worst-case execution scenarios, which we called Worst-Case Energy Consumption (WCEC) prediction. We used SPLs as a case study, implementing the technique in a prototype tool and evaluating it's accuracy by performing an experiment using existing SPLs.

The energy-aware evolution topic was addressed in two ways. First, we studied the energy impact of replacing energy-greedy coding patterns with less greedy alternatives. We did so by performing a large-scale study, using hundreds of Android applications and testing them in several different scenarios. This allowed us to find statistical evidence of what patterns provide the most significant savings when replaced, and how replacing multiple patterns can affect such savings.

Finally, building on the previous study, we developed a new concept, called Energy Debt, which goal is to help developers assess the impact of addressing (or retaining) energy-greedy patterns in their code, in the short and in the long term, by considering a software's different releases. This concept is extensible to any target system and language, and we performed an initial experiment using an Android application to properly explain it and motivate it.

We strongly believe that this thesis, in addition to addressing the existing lack of energy related knowledge and tools, also provides novel and promising techniques and findings, which we hope can be used by other researchers in the area to continue exploring these subjects.

Resumo

Apoiar Desenvolvedores de Software na Tomada de Decisões para Redução de Energia

Na última década, a análise e melhoria do consumo energético em software tem-se estabelecido como uma nova preocupação chave para os desenvolvedores. Porém, como estudos recentes demonstram, os desenvolvedores ainda mostram dificuldades em entender como analisar e melhorar a eficiência energética do seu código. Cientes deste desafio, a comunidade académica tem continuamente trabalhado para providenciar novas ferramentas, técnicas, e resultados em que se possa confiar.

Com esta tese, o foco principal é estender ainda mais o conhecimento e mecanismos disponíveis para os desenvolvedores poderem raciocinar de forma estática sobre a eficiência energética do seu código. As nossas contribuições podem ser agrupadas em dois tópicos: previsão do comportamento energético e evolução orientada à redução de energia.

Para abordar a previsão de energia, desenvolvemos uma nova abordagem que combina análise estática de programas e modelos de consumo energético, com o objetivo de providenciar estimativas precisas de consumos em cenários de pior caso de execução, ao qual chamamos de Previsão de Consumo Energético no Pior Caso. Usamos SPLs como caso de estudo, e implementamos esta técnica num protótipo e avaliamos a sua precisão através de um estudo usando SPLs já existentes.

O tópico de evolução orientada à redução energética foi abordado de duas formas. Primeiro, foi estudado o impacto inerente à substituição de padrões de código, conhecidos por serem energeticamente ineficientes, por alternativas mais viáveis. Isso foi feito através de de um estudo em larga escala, usando centenas de aplicações Android e testando-as em diferentes cenários. Isto permitiu encontrar evidência estatística de quais os padrões que, quando substituídos, produzem maiores ganhos, mas também perceber como múltiplas substituições afetam esses ganhos.

Por fim, e sustentando-nos no estudo anterior, desenvolvemos um novo conceito, chamado Débito Energético, cujo objetivo é ajudar os desenvolvedores a avaliar o impacto de substituir (ou manter) no seu código padrões reconhecidamente ineficientes do ponto de vista energético, quer a curto quer a médio/longo prazo, através da análise combinada das diferentes versões de um software. Este conceito é genérico e extensível para qualquer sistema ou linguagem, e para o explicar e motivar adequadamente efetuamos um estudo inicial usando uma aplicação Android.

É nossa convicção de que esta tese, além de endereçar a questão relativa à falta de conhecimento e de ferramentas relacionados com análise de energia, também fornece novas e promissoras técnicas e descobertas, das quais esperamos que possam ser usados por outros investigadores nesta área.

Contents

1 Introduction **1**

1.1 Problem Statement and Research Goals 5

1.2 Thesis Structure & Origin of Chapters 6

1.3 Further Contributions 8

2 State of the Art **13**

2.1 Energy Profiling 14

2.2 Detection of Energy-Greedy Code 18

2.3 Adjusting Software and Improving Energy 21

2.4 Energy Behavior Prediction 25

2.5 Additional Energy Optimization Approaches 27

3 Energy Estimations in Software Product Lines **31**

3.1 Introduction 32

3.2 Static Analysis in SPLs 35

3.2.1 Basic Concepts of Software Product Lines 35

3.2.2 Static Dataflow Analysis concepts 36

3.2.3 Static Analysis in Software Product Lines 39

3.3 Static Energy Analysis in SPLs 40

3.3.1 Static Prediction of Energy Behavior 42

3.3.2 Data-Flow Analysis 43

3.3.3 Combining SPL Static Analysis With Energy Estimation 45

3.3.4 Worst Case Prediction 46

3.4 Overview of the Prototype Tool 49

3.5 Experimental Evaluation 50

3.5.1 Static Model Creation 52
3.5.2 Experimental Methodology 53
3.5.3 Accuracy Enhancements 55
3.6 Threats to Validity 60
3.7 Conclusions 61
4 Energy Code Smells in Android **63**
4.1 Introduction 64
4.2 EGAPs - Energy Greedy Android Patterns 66
4.2.1 EGAP #1 - `Draw Allocation` 67
4.2.2 EGAP #2 - `Wakelock` 68
4.2.3 EGAP #3 - `Recycle` 68
4.2.4 EGAP #4 - `Obsolete Layout Parameter` 69
4.2.5 EGAP #5 - `View Holder` 70
4.2.6 EGAP #6 - `HashMap Usage` 71
4.2.7 EGAP #7 - `Excessive Method Calls` 71
4.2.8 EGAP #8 - `Member Ignoring Method` 71
4.2.9 EGAPs #9, #10 & #11 - `Resource Leak` 72
4.3 The Android Application Repository 72
4.4 Automatic EGAP Detection And Transformation On Android Code 73
4.4.1 Detection Methodology 74
4.4.2 Automatic Refactoring Approach 75
4.4.3 Results and Discussion 75
4.5 Analyzing The Energy Impact Of Refactoring EGAPs 76
4.5.1 Experimental Setup 77
4.5.2 Experimental Pipeline 78
4.5.3 Results and Discussion 79

4.6 Threats To Validity 84
4.7 Conclusions 85

5 Towards Combining Energy Smells with Decision Support 87

5.1 Introduction 88
5.2 Introducing Energy Debt Concepts 89
5.2.1 Concept Overview 89
5.2.2 Counting Expenses & Estimating Debt 92
5.2.3 Paying Interests 96
5.3 Preliminary Experiment 98
5.3.1 The Experimental Software System 98
5.3.2 Energy Debt Analysis 99
5.4 Conclusions 102

6 Multi Purpose Tools for Inspecting Energy of Programs 103

6.1 Introduction 104
6.2 C-LEM: Energy Model Creator for the CIL language 104
6.3 Serapis: Estimate WCEC through Program Analysis and Energy Models 108
6.4 LintED: A Lint-based EGAP Detector 112
6.5 Chimera: An EGAP-oriented Testing Tool for Android 115

7 Conclusions 119

7.1 Future Work 122

References 123

A Chapter 3 Appendix 139

A.1 Full Study Results 139

B Chapter 4 Appendix 142

B.1 Energy Consumption Distribution: Before and After Refactoring EGAPs 142

B.2 Mann–Whitney U and Cohen's d Results 165

Acronyms

AST Abstract Syntax Tree

CFG Control Flow Graph

CIL C Intermediate Language

CPU Central Processing Unit

DSPL Dynamic Software Product Line

ED Energy Debt

EGAP Energy-Greedy Android Pattern

GoF Gang of Four

ICT Information and Communication Technology

IDE Integrated Development Environment

ILP Integer Linear Programming

IoT Internet of Things

IPET Implicit Path Enumeration Technique

JCF Java Collection Framework

LLVM Low Level Virtual Machine

MVC Model-View-Controller

MVP Model-View-Presenter

OS Operative System

PMIC Power Management Integrated Circuit

RAPL Running Average Power Limit

SDK Software Development Kit

SPL Software Product Line

UI User Interface

WCEC Worst Case Energy Consumption

WCET Worst Case Execution Time

List of Figures

1.1 Thesis Outline and Chapter Flow 6

2.1 Monsoon Usage Scheme 14

2.2 Reported Accuracy for RAPL and Trepn 17

2.3 The Output from Battery Historian 18

2.4 How Energy is Consumed Inside Real-World Android Apps (obtained from [Li et al., 2014a]) 19

2.5 The Energy Cost on Including Ads in Android Apps (obtained from [Gui et al., 2015]) 20

2.6 Energy Consumed by Standard Operations on JCF collections (obtained from [Hasan et al., 2016]) 23

2.7 Energy Impact of Strict vs Lazy Evaluation on Haskell Data Structures (obtained from [Melfe et al., 2018]) 24

2.8 Energy Savings (in %) Obtained From Bundling HTTP Requests. (obtained from [Li et al., 2016]) 27

2.9 Energy Impact of Bundling/Dropping Events in 4 Android Applications. (obtained from [Chowdhury et al., 2019]) 28

2.10 Energy Footprint of UI Testing Frameworks: swipe and long tap tasks (obtained from [Cruz and Abreu, 2019b]) 29

3.1 Control-Flow Graph for the example in Listing 3.1 37

3.2 Lattice for Sign Analysis 38

3.3 Whole-program transfer function and sign analysis (as fixed point computation) for the example in Figure 3.1 39

3.4 Sign analysis' (partial) result, performed for the example in Figure 3.1, using Simultaneous Feature-Sensitive 41
3.5 Example 3.1 data-flow analysis (loop's first iteration.) 44
3.6 Calculating local energy bounds for each node in the CFG 46
3.7 IPET constraints for product $\{A\}$ of Example 3.1 48
3.8 The Serapis Workflow 49
3.9 Accuracy comparison: Original model vs Simulated annealing-improved model 59

4.1 Case Study Pipeline 79
4.2 Overview: Experimental Results 80
4.3 Gains Comparison: EGAPs vs Combinations 82

5.1 Technical Debt vs Energy Debt Terminology 90
5.2 Energy Debt Thresholds Increase Over Time 92
5.3 Estimating Energy Debt per Release 94
5.4 Accumulation of Interest 97
5.5 Minimum and Maximum Energy Debt for EscapeApp releases 100

6.1 Workflow of C-LEM: the Energy Model Creator 105
6.2 Overview of the Preprocessing and CFG Creation Steps 110
6.3 Chimera's Architecture 115

B.1 Energy Before/After Refactoring. **EGAPs**: `DrawAllocation` (DA) 142
B.2 Energy Before/After Refactoring. **EGAPs**: `Wakelock` (WL) 143
B.3 Energy Before/After Refactoring. **EGAPs**: `Recyle` (R) 143
B.4 Energy Before/After Refactoring. **EGAPs**: `ObsoleteLayoutParam` (OLP) . 144
B.5 Energy Before/After Refactoring. **EGAPs**: `ViewHolder` (VH) 144
B.6 Energy Before/After Refactoring. **EGAPs**: `HashMapUsage` (HMU) 145

B.7 Energy Before/After Refactoring. **EGAPs**: ExcessiveMethodCalls (EMC) . 145
B.8 Energy Before/After Refactoring. **EGAPs**: MemberIgnoringMethod (MIM) . 146
B.9 Energy Before/After Refactoring. **EGAPs**: DA, EMC 146
B.10 Energy Before/After Refactoring. **EGAPs**: DA, HMU 147
B.11 Energy Before/After Refactoring. **EGAPs**: DA, MIM 147
B.12 Energy Before/After Refactoring. **EGAPs**: R, OLP 148
B.13 Energy Before/After Refactoring. **EGAPs**: R, HMU 148
B.14 Energy Before/After Refactoring. **EGAPs**: R, MIM 149
B.15 Energy Before/After Refactoring. **EGAPs**: OLP, VH 149
B.16 Energy Before/After Refactoring. **EGAPs**: OLP, HMU 150
B.17 Energy Before/After Refactoring. **EGAPs**: OLP, EMC 150
B.18 Energy Before/After Refactoring. **EGAPs**: OLP, MIM 151
B.19 Energy Before/After Refactoring. **EGAPs**: VH, HMU 151
B.20 Energy Before/After Refactoring. **EGAPs**: VH, EMC 152
B.21 Energy Before/After Refactoring. **EGAPs**: VH, MIM 152
B.22 Energy Before/After Refactoring. **EGAPs**: HMU, EMC 153
B.23 Energy Before/After Refactoring. **EGAPs**: HMU, MIM 153
B.24 Energy Before/After Refactoring. **EGAPs**: EMC, MIM 154
B.25 Energy Before/After Refactoring. **EGAPs**: DA, HMU, EMC 154
B.26 Energy Before/After Refactoring. **EGAPs**: DA, HMU, MIM 155
B.27 Energy Before/After Refactoring. **EGAPs**: DA, EMC, MIM 155
B.28 Energy Before/After Refactoring. **EGAPs**: R, OLP, HMU 156
B.29 Energy Before/After Refactoring. **EGAPs**: OLP, VH, HMU 156
B.30 Energy Before/After Refactoring. **EGAPs**: OLP, VH, EMC 157
B.31 Energy Before/After Refactoring. **EGAPs**: OLP, VH, MIM 157
B.32 Energy Before/After Refactoring. **EGAPs**: OLP, HMU, EMC 158

B.33 Energy Before/After Refactoring. **EGAPs**: OLP, HMU, MIM 158

B.34 Energy Before/After Refactoring. **EGAPs**: OLP, EMC, MIM 159

B.35 Energy Before/After Refactoring. **EGAPs**: VH, HMU, EMC 159

B.36 Energy Before/After Refactoring. **EGAPs**: VH, HMU, MIM 160

B.37 Energy Before/After Refactoring. **EGAPs**: VH, EMC, MIM 160

B.38 Energy Before/After Refactoring. **EGAPs**: HMU, EMC, MIM 161

B.39 Energy Before/After Refactoring. **EGAPs**: DA, EMC, HMU, MIM 161

B.40 Energy Before/After Refactoring. **EGAPs**: OLP, VH, HMU, EMC 162

B.41 Energy Before/After Refactoring. **EGAPs**: OLP, VH, HMU, MIM 162

B.42 Energy Before/After Refactoring. **EGAPs**: OLP, VH, EMC, MIM 163

B.43 Energy Before/After Refactoring. **EGAPs**: OLP, HMU, EMC, MIM 163

B.44 Energy Before/After Refactoring. **EGAPs**: OLP, VH, HMU, EMC, MIM 164

List of Tables

2.1 Language Rankings for Different Optimization Goals (obtained from [Pereira et al., 2017b]). 22

3.1 Overview of SPLs used in the experiment 51

3.2 Study Results 54

4.1 Applications Repository Metrics 73

4.2 Results Of EGAP Detection And Refactor Phases 77

4.3 Results of Mann–Whitney U test for individual combinations 81

5.1 Energy Debt Smell Catalog 93

5.2 Detected smells on EscapeApp releases 99

A.1 Study Results for San Diego Benchmark SPL 139

A.2 Study Results for Elevator SPL 139

A.3 Study Results for EmailSystem SPL 140

A.4 Study Results for Minepump SPL 141

B.1 Mann-Whitney U results: `DrawAllocation` 165

B.2 Mann-Whitney U results: `Recycle` 165

B.3 Mann-Whitney U results: `ObsoleteLayoutParam` 166

B.4 Mann-Whitney U results: `ViewHolder` 166

B.5 Mann-Whitney U results: `HashMapUsage` 167

B.6 Mann-Whitney U results: `ExcessiveMethodCalls` 167

B.7 Mann-Whitney U results: `MemberIgnoringMethod` 168

1 Introduction

Since the appearance of the first computers, software development has been adapted several times, if not reinvented. There has always been an increasing demand for software to offer new and powerful features and to run in different host systems. This challenging demand has made software development into a thorough and structured discipline, where concepts such as modularity and maintainability were the key. The process of adding new features (or functional properties) was turned into a more methodic and organized process.

Software functional properties, however, were not the only ones to take into consideration. As the demand for highly complex and powerful software increased, so did the necessity for performance. Hence, the non-functional properties of software appeared as new concerns in the software development discipline. Several concepts/metrics, such as computational effort, time complexity, memory and CPU usage, were initially considered as a way to describe software performance. Hence, understanding how functional properties affect the non-functional ones, by accurately measuring or estimating values for these metrics, was an important aspect for developers, which relied on tools such as multi purpose profilers, like Perf [Per, 2018] and Valgrind [Val,], to assess the non-functional properties of their software.

The devices targeted for running software kept continued to evolve, and the end users themselves developed new requirements and expectations towards what software should do and how it should behave. These aspects influenced the software development discipline, and a notorious turning point came with the emergence and widespread of mobile devices. With "mobile devices" we consider all devices capable of running on a battery-sustained hardware (i.e. from smartphones to embedded and IoT devices). The battery uptime is seen as the major usage limitation of these devices. Users cannot rely totally on it, as its capacity is limited, and trading off performance for uptime is often an employed strategy, in order to increase the battery duration, which is the major usage limitation of these devices. With this concern in mind, a new performance issue has raised attentions of both researchers and developers: energy efficiency [Hooper, 2008, Harmon and Auseklis, 2009]. Running software implies the consumption of energy, and although the hardware is what actually consumes it, the software is the trigger which ultimately

decides on how the hardware operates. Hence, improvements performed at the software level can propagate and make the hardware consume less energy [Harmon and Auseklis, 2009, Costa-Requena et al., 2014].

Although the interest in improving the energy efficiency of software is understandably higher when considering mobile devices, it should not be neglected in other areas. Reducing energy can also result in reducing monetary costs (since energy costs money), and it also helps the sustainability of the environment (due to the reduction of CO2 emissions). As recent reports show, the Information and Communication Technology (ICT) area is responsible for up to 9% of the worldwide consumption, which in turn is mostly supported by unsustainable sources [Ene, 2019]. The energy consumed by datacenters alone is estimated to represent around 1% of worldwide consumption [Masanet et al., 2020]. Employing energy saving practices to software development can have a great deal of impact in a wide variety of contexts. For instance, the smallest energy improvement in software that runs on several instances in data centers, which have been consistently growing their electricity use during the last decades [Koomey, 2011], would ultimately result in consistent money savings and reduction of CO2 emissions [Rda, 2017], and increasing the uptime of several IoT devices, for as little as 1 minute per day, eventually results in a higher amount of collected data.

Aware of the importance of reducing the energy footprint in the ICT sector, developers then started to reveal new concerns, regarding how to write code in an energy-efficient manner [Pinto et al., 2014, Pang et al., 2016]. These concerns were increased once end users started to reveal being more interested in energy efficient applications, specially when considering battery-powered devices [Ath, 2017, Fri, 2019]. To tackle this challenge, researchers have been working, for the last decade, on different approaches to analyze and reduce the energy consumed by a software system. Essentially, those works have focused on how to provide answers to 4 main questions in this area:

1. how can energy be accurately monitored?
2. how to reason about consumption to detect energy efficient/inefficient code?
3. can the energy consumption in software be accurately predicted?
4. how can developers improve/evolve software with energy inefficient code, in an energy-aware manner?

A wide variety of works appeared to address these questions. New monitor tools, commonly

referred to as energy profilers, were created, differing in the way they collect data (either using hardware [Peterson et al., 2011] or software based mechanisms [Pandruvada, 2014, David et al., 2010, Di Nucci et al., 2017b, Hao et al., 2012, Li et al., 2013]), and on their target systems (e.g., Android devices [Di Nucci et al., 2017b, Hao et al., 2012, Li et al., 2013] or desktop applications [Pandruvada, 2014]). Using these tools, researchers could understand how the monitored energy relates to certain code blocks, which means they could reason about energy consumption within software systems.

Reasoning about software energy consumption ultimately means understanding if a certain consumption value can be considered excessive, and what in the code is causing it. Given the high complexity and diversity of software nowadays, the fact is that several factors can be the cause of energy leaks, and to identify them it requires a dynamic approach, based on testing and observation. Different approaches were proposed by researchers on this topic, which included new techniques for detecting energy leaking methods [Couto et al., 2014, Pereira, 2018, Pereira, 2018], comparing the energy efficiency of different data structures [Hasan et al., 2016, Pereira et al., 2016, Pinto and Castor, 2014, Manotas et al., 2014], APIs [Linares-Vásquez et al., 2014], or design patterns [Sahin et al., 2012, Feitosa et al., 2017], and exploring the effects of code transformation techniques (such as refactorings [Sahin et al., 2014a, Verdecchia et al., 2018, Cruz and Abreu, 2018, Couto et al., 2020a] and code obfuscation [Sahin et al., 2014b]). This way, researchers were able to understand what code blocks (or coding practices) could be classified as energy bugs, and how to tackle them by providing more suitable and less energy-greedy alternatives.

In some cases, measuring and reasoning about energy can be infeasible. If we consider, for instance, software families, where several sub-systems share most of their code but have quite different usage contexts, the reasoning for one of those systems may not be applicable to others. Moreover, depending on how extensive the family is, monitor each and every sub-system may be an unrealistic approach. As such, researchers also explored the advantages of static approaches. Hence, new techniques were developed with the goal of inferring how energy consumption varies in software, rather than just measure it. Using program analysis to estimate the consumption of specific execution scenarios [Hao et al., 2013, Hao et al., 2012], model-checking energy behavior [Nakajima, 2015a, Nakajima, 2015b], and checking the occurrence of energy-greedy APIs/code patterns [Jabbarvand et al., 2015], are some examples of such techniques.

Regardless of the followed analysis approach (static or dynamic), the ultimate goal is always to provide developers with knowledge on how to make their software more energy efficient. With this thesis, we seek to further extend this knowledge, by providing additional contributions to two energy-related research topics: energy estimation, and energy-aware software evolution. Regarding estimation, we describe an approach that, through static program analysis mechanisms, can accurately estimate the amount of energy consumed by software in a worst-case scenario. Here, we focused on Software Product Lines (SPL), a well established software development technique, as our case study. We managed to validate this approach by using examples that have been widely explored in the SPL research area.

To address energy-aware software evolution, we first conducted a study to assess the energy impact of transforming (i.e. refactoring) individual code blocks (i.e. patterns), known to be energy inefficient. For this purpose, we first searched in the literature for studies where such energy-greedy patterns were identified, and we were able to collect 11 of them. All such patterns were tested under the same conditions, within a large-scale set of applications where they occur, and we were able to understand which patterns produce higher energy savings when replaced/transformed, and how the gainings vary when applying more than one transformation at a time. Here, we focused on Android-related patterns, as the literature is quite extensive and a larger number of patterns were studied.

The information obtained from the aforementioned study can be tricky to understand by developers. With the aim of providing the information more comprehensibly, we started developing a new concept called Energy Debt. This concept considers successive versions of an application (i.e., its releases), and the reported energy-greedy patterns on each one. With this, Energy Debt creates a new metric to relate the known energy gains of each pattern with the application's uptime. Hence, it is clearer to understand how much can the energy savings vary across releases and depending on how the application is used. It also allows practitioners to consider the impact of neglecting energy faulty code for too long.

With the contents of this thesis, we seek to further expand the knowledge that practitioners have on how to improve their code in a more energy efficient manner. We also developed new techniques, implemented in prototype tools, which are targeted both for developers to analyze their code and for other researchers to continue extending the knowledge on how to build more energy efficient code.

1.1 Problem Statement and Research Goals

The challenge of understanding the impact that software systems have on energy consumption can be tackled through multiple approaches. Nevertheless, if the main goal is to determine how to reduce energy through software improvements, the ultimate target must be how to provide developers structured and trustworthy information on how to do it. Recently, it has been proved that developers lack the knowledge and the adequate tools to aid them on this task [Pinto and Castor, 2017, Pang et al., 2016].

Hence, with this thesis, we focus on addressing this main research question:

Main Research Question

How can we increase the knowledge that is available to practitioners, so that they can accurately assess and improve the energy efficiency of their programs/applications?

This research question covers a wide variety of topics, and thus it can be addressed in multiple ways. As such, with the contributions incorporated in this thesis, we address this question by exploring the following research goals:

Research Goal 1: To accurately being able to predict the energy consumed by several similar software systems (Chapter 3).

Energy prediction combines static program analysis techniques with energy modeling, to estimate the energy behavior of a program. When analyzing multiple programs that share most of their code, this approach can reduce the analysis effort. We focused on developing an approach that can report accurate estimations for multiple programs within the same software family (e.g., a Software Product Line), assuming a worst-case execution scenario.

Research Goal 2: To understand how documented energy improvements behave when applied together, and in a wide variety of contexts (Chapter 4).

Energy greedy code is a widely explored topic, particularly for Android applications. Nevertheless, although several works studied the benefits of using more energy efficient alternatives, this was often tested individually and using a small set of applications/execution scenarios. Thus, we performed a large scale study where the energy impact of several Android code patterns (found in

the literature) was assessed. We also sought to understand if the energy impact of such patterns is affected by "fixing" several of them at the same time.

Research Goal 3: To develop an approach which can relate energy analysis/improvement with software evolution and maintenance (Chapter 5).
Locating energy-greedy code provides valuable information for practitioners on how to improve their software. Fixing them, however, is sometimes an arduous task that involves (for instance) dealing with old or complex code, or making the code less readable/maintainable. As such, we strongly believe that the information regarding energy-greedy code should be complemented with estimations that can reflect the long-term implications of not addressing these "energy bugs". Hence, we developed an approach that helps practitioners understand how leaving energy-greedy code unattended for multiple releases of their software can have significant implications on the energy consumption.

Each of the aforementioned research goals can be further expressed using context-specific research questions. Hence, in each chapter related to one of the aforementioned research goals, we will introduce those questions and provide an answer to each one.

1.2 Thesis Structure & Origin of Chapters

The outline of this thesis is depicted in Figure 1.1, which also suggests possible flows for reading the remaining chapters. Chapters aligned vertically can be read independently.

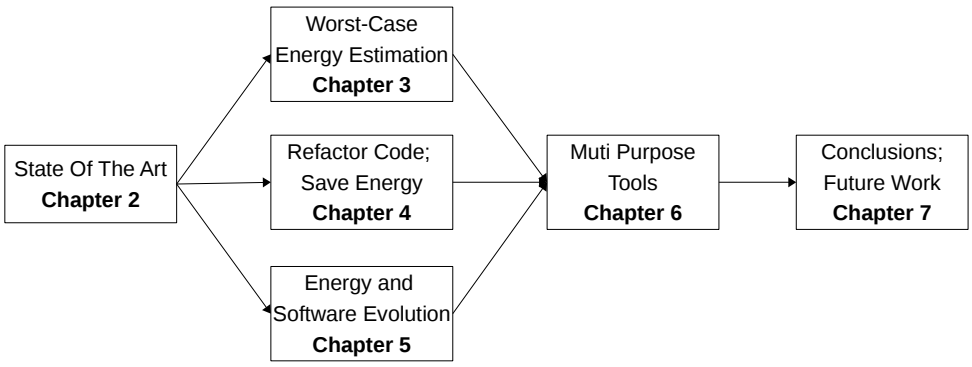


Figure 1.1: Thesis Outline and Chapter Flow

We start by presenting, in Chapter 2, a thorough analysis and discussion of the state of the art directly related to this thesis research topics.

The 4 chapters following the state of the art contain the fully detailed description of out core contributions, which resulted from the work developed in this thesis. The contents of such chapters have either been published in peer-reviewed journals and conferences, or are currently under review. Next, we summarize the contents of each one, while also indicating the publication(s) that originated them.

Chapter 3 describes our approach for predicting energy consumption, in a worst-case scenario, for software products within a SPL. This is based on the following publications:



"Products go green: Worst-case energy consumption in software product lines"
Marco Couto, Paulo Borba, Jácome Cunha, João P. Fernandes, Rui Pereira, João Saraiva
In: 21st International Systems and Software Product Line Conference, SPLC, 2017
pages: **84 – 93** | DOI: **10.1145/3106195.3106214**



"Statically analyzing the Energy Efficiency of SPLs"
Marco Couto, João P. Fernandes, João Saraiva
Submitted to: IEEE Transactions on Sustainable Computing, 2020
16 pages long

Chapter 4 describes our large-scale study where we aim at understanding how different energy-greedy code patterns affect the energy consumption of Android applications, when considered alone and when combined with others. This is based on the following publications:



"Energy Refactorings for Android in the Large and in the Wild"
Marco Couto, João Saraiva, João P. Fernandes
In: 27th IEEE International Conference on Software Analysis Evolution and Reengineering, SANER, 2020
pages: **217 – 228** | DOI: **10.1109/SANER48275.2020.9054858**

Part of the aforementioned study was also included in a book chapter publication. Here, our observations are presented in a teaching perspective, i.e., using a real-world example we explain what makes a code pattern energy-greedy, and how to use a less greedy alternative:



"Patterns and Energy Consumption: Design, Implementation, Studies and Stories"
Daniel Feitosa, Luís Cruz, Rui Abreu, João P. Fernandes, Marco Couto and João Saraiva
Submitted to: Software Sustainability, 2020
32 pages long

Chapter 5 contains the definition of a new concept, Energy Debt, which aims at providing more targeted and suited information for developers, regarding the state of their software releases when energy-greedy patterns exist in the code. This concept was developed in co-authorship with Daniel Maia, a MSc student, and is based on the following publications:



"On Energy Debt: Managing Consumption on Evolving Software"
Marco Couto, Rui Pereira, Daniel Maia, João Saraiva
In: International Conference on Technical Debt, TechDebt, 2020
5 pages long (to appear)

Chapter 6 contains a more in-depth description of all the tools developed in the context of the aforementioned publications.

Finally, we present the conclusions of our work, and discuss some future work possibilities, which are included in Chapter 7.

1.3 Further Contributions

The main contributions of this thesis, which are introduced and thoroughly explained in the following chapters, are the ones directly related to the thesis core subject: energy saving oriented decision support in software development. In other words, providing methodologies and tools that support the analysis and evolution of software, when the key concern is reducing energy consumed over time.

The topic of the thesis, however, is included in a broader research area, which is commonly referred to as **Energy Analysis in Software** (also known as **Green Computing**). During the time period on which this thesis was produced, we also contributed directly to several other research works on the main area, by formulating the initial concept, actively working in the studies invoved, and in the writing of the article. The scope and goals of each of those individual works

vary considerably. Next, we will present a small description for each of these works, along with the publications that were originated from them.

Energy Efficiency of Programming Languages

Programming languages differ among them in a number of ways. From their grammar to their programming or execution paradigm, all the traits of a language can ultimately affect the performance aspects of their programs, such as execution time or memory management.

Given this assumption, we designed a study where the energy efficiency aspect of languages was put into comparison. A series of benchmark programs, obtained from the Computer Language Benchmark Game (CLBG)[1], was used as comparison test subjects for the languages. As the benchmarks were implemented in several general purpose languages (such as C, Python or Java), the energy collected in each test/language allowed understanding how different languages or programming/execution paradigms can affect energy.

This work resulted in the following two publications:

"Towards a Green Ranking for Programming Languages",
Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, João Saraiva
in 21st Brazilian Symposium on Programming Languages (SBLP 2017) **(best paper)**.
pages: **1 – 8** | DOI: **10.1145/3125374.3125382**

"Energy Efficiency Across Programming Languages: How do Energy, Time, and Memory Relate?",
Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João P. Fernandes, João Saraiva
in 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017).
pages: **256 – 267** | DOI: **10.1145/3136014.3136031**

GreenHub - Android Energy Analysis Through Collaborative Data

Understanding the impact that mobile applications' design and implementation has on energy consumption, whether to compare them with others or to improve them, is an aspect that concerns to developers. It is possible, however, to take the energy analysis to a higher abstraction level, so that end-users can also understand how the energy drain on their mobile device can be

[1] CLGB page: https://benchmarksgame-team.pages.debian.net/benchmarksgame (visited on October 19, 2020).

reduced.

GreenHub[2] is a step forward towards helping Android users understand how their whole device is draining energy, and how can it be reduced. Through its infrastructure, it collects all sorts of energy related usage data from real-words Android devices, on a daily basis. The data is provided to the research community, which enables a whole new approach for researchers to study what usage patterns end-users have that causes energy to drain faster, and can pass that information on to all Android end-users.

This work resulted in the following two publications:

"GreenHub Farmer: Real-world Data for Android Energy Mining",
Hugo Matalonga, Bruno Cabral, Fernando Castor, Marco Couto, Rui Pereira, Simão Melo de Sousa, João P. Fernandes
in IEEE/ACM 16th International Conference on Mining Software Repositories (MSR 2019)
pages: **171 – 175** | DOI: **10.1109/MSR.2019.00034**

"GreenHub: A Large-Scale Collaborative Dataset to Battery Consumption Analysis of Android Devices",
Rui Pereira, Hugo Matalonga, Marco Couto, Fernando Castor, Bruno Cabral, Pedro Carvalho, Simão Melo de Sousa, João P. Fernandes
in Empirical Software Engineering (submitted)
61 pages long

GreenSource - Repository of Android Energy Metrics

When the goal is to detect energy-greedy behavior or code in Android applications, researchers often need to build and run energy tests. This can obviously be a costly and time consuming task.

The goal of the GreenSource repository is to reduce the need of building energy tests. Using a large collection of open-source Android applications as test subjects, this repository contains a large-scale compilation of test metrics. These metrics were obtained by running several energy tests on each application, using different Android devices and testing frameworks.

There is an extensive list of considered metrics, which include energy consumed, list of method calls, sensors usage, or even CPU load. This information can be used for information retrieval purposes, in order to relate energy to source code or applications' behavior.

[2]GreenHub webpage: https://greenhubproject.org/ (visited on October 19, 2020)

This work resulted in the following publication:

"GreenSource: a Large-scale Collection of Android Code, Tests and Energy Metrics",
Rui Rua, Marco Couto, João Saraiva
in IEEE/ACM 16th International Conference on Mining Software Repositories (MSR 2019)
pages: **176 – 175** | DOI: **10.1109/MSR.2019.00035**

How Memoization Affects Energy Consumption

Several aspects related to software implementation, such as the language or data structures chosen, can affect energy consumption. Similarly, existing techniques designed to improve performance aspects of software might also play a role in helping reduce energy.

One such example is memoization. With this work, the focus was to understand if applying memoization mechanisms to compatible methods can in fact be used as an energy optimization technique. The initial results, obtained from experiments performed in Android applications, show that there are several scenarios where memoization can help in reducing energy.

This work resulted in the following publication.

"Towards Using Memoization for Saving Energy in Android",
Rui Rua, Marco Couto, Adriano Pinto, Jácome Cunha, João Saraiva
in 22nd Ibero-American Conference on Software Engineering (CIbSE 2019)
pages: **279 – 292** |

Energy Leak Localization

One of the biggest challenges when analyzing energy in software is to determine where can the potential energy-greedy code (or energy leaks) be located. The amount of energy spent by a code block (for instance, a method) can be notoriously high, but completely necessary.

In order to address this challenge, a technique was proposed, called SPELL, to detect energy leaks in Java applications. It considers the spectrum of execution traces, obtained from tests runs, to detect the code blocks that have the most influence in the application's overall consumption. This helps developers understand where they can start looking at in their code, in order to optimize energy consumption.

This work resulted in the following two publications:

"Helping Programmers Improve the Energy Efficiency of Source Code",
Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João P. Fernandes, João Saraiva
in IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C 2017)
pages: **238 – 240** | DOI: **10.1109/ICSE-C.2017.80**

"Spelling out Energy Leaks: Aiding Developers Locate Energy Inefficient Code",
Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João P. Fernandes, João Saraiva
in Journal of Systems and Software 2020, Volume 161
pages: **110463** | DOI: **10.1016/j.jss.2019.110463**

Energy Efficiency in Android Virtual Keyboards

One of the core applications included in every distribution of the Android OS is the virtual keyboard application. As opposed to traditional mechanical ones, this keyboard is purely a software system. As such, depending on the implementation, it can provide more or less efficient features.

Keyboards are fairly comparable among them, since they offer the same core features. With this work, five widely used keyboard applications in Google Play where put into comparison, in terms of the energy efficiency of their identical features.

This work resulted in the following publication:

"Greenspecting Android Virtual Keyboards",
Rui Rua, Tiago Fraga, Marco Couto, João Saraiva
in 7th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2020)
(to appear)
11 pages long

2

State of the Art

Energy analysis in software has become a very broad and prolific research area over the last decade. With the widespread usage of battery-powered devices of all sorts, and the resulting growing interest from developers on the subject, researchers have been proposing different techniques, tools and methodologies for a more sustained energy-aware development. Accurately profiling and predicting energy consumption, along with understanding how to reduce energy through software improvements and giving concrete feedback to developers have been the major concerns.

In this chapter, we present an overview of several research works, related to energy analysis and optimization in software. The extent of these research works covers a wide variety of subjects, and occasionally have overlapping concepts and approaches. Hence, we divided them into different sections, according to their main goals and contributions.

In Section 2.1, we discuss energy monitoring tools/techniques that have been proposed, their differences and usage contexts. After describing such mechanisms, we look into how other research works used them to detect potential energy leaks in the code of various software systems, which is described in Section 2.2. In Section 2.3 we then explain the works that focused on studying how energy can be influenced by different software implementation decisions, such as the selected language, design patterns, or data structures. We also look into techniques suggested for estimating energy consumption in controlled contexts, which is included in Section 2.2. Finally, we include in Section 2.5 a description of other energy optimization techniques that have been suggested in a more broad context.

2.1 Energy Profiling

Over the years, researchers been have proposing different approaches to accurately measure the energy consumed by software systems. The initial lack of appropriate tools and knowledge, and the fact that operating systems were not yet tailored to provide energy-related information in the same way they did for execution time or memory usage, were challenges to overcome.

Without the existence of any purely software-based mechanism, researchers explored approaches based on external measurement devices, the so-called power monitors. The main advantage of this approach was that the obtained values were not estimations, but actual observed measurements. An example of such a device is the Monsoon power monitor[1]. This device is connected directly to the battery power connectors of a standard lithium ion battery-powered device (e.g. smartphones), as depicted in Figure 2.1. Since it acts as a replacement for the battery, providing power to the device, it knows exactly the amount required at any given time, and depending on the model it can collect up to 5000 measurements per second.

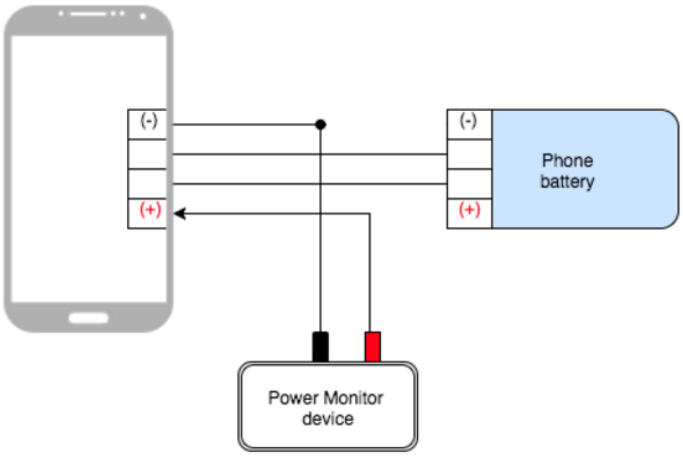


Figure 2.1: Monsoon Usage Scheme

Requiring a power monitor such as Monsoon to monitor consumption can sometimes be an inconvenient. The additional cost of acquiring the device can be impractical for some, and if the goal it to aid developers it is inappropriate to require them to have one such device. Motivated by this downside, researchers focused on building energy profilers, which could be purely software-based solutions, and provide accurate estimations, rather then observed measurements. This was initial achieved by combining the values reported by power monitors, with hardware properties

[1] Monsoon webpage: https://www.msoon.com/powermonitor-support (visited on October 19, 2020).

that could be obtained at runtime (such as CPU load, screen brightness, or number of memory accesses). Hence, the notion of energy models was created.

The approach based on energy models was followed by some works, which focused on building profilers targeted for Android devices. **PowerTutor** [Zhang et al., 2010], **DevScope** [Jung et al., 2012], or **AppScope** [Yoon et al., 2012] are some examples of such works, each one with different goals. All the three report the amount of energy consumed by each hardware component. Yet, PowerTutor is able to divide that consumption per active applications, while DevScope's main feature is a technique for dynamically and independently updating the model during profiling, making it more accurate. The strategy employed by DevScope was also followed by AppScope, which ended up being the same type of profiler as PowerTutor (i.e. application-level profiler), but with a substantial increased accuracy. The **PowerProf** [Kjærgaard and Blunck, 2012] and **eProf** [Pathak et al., 2011, Pathak et al., 2012] profilers are also similar examples, although the context for which they were initially developed is more specific than the previous referred profilers.

Using these energy models allowed the aforementioned profilers to account for hardware state changes, which could have a significant impact on energy consumption. Yet, monitoring these variations requires the continuous fetching and processing of state data, which implies that their sampling rate (i.e., the number of estimations per time unit) was quite low[2]. Hence, when the goal is to analyze consumption at a more fine-grained granularity level (such as methods or lines of code), these tools simply were not adequate.

A different approach for energy models was deemed necessary. As such, researchers explored alternatives to trade-off a more detailed information for an analysis with finer granularity level, while keeping accurate estimations. In order to do this, two strategies were employed. The first one consists of creating an instruction-level energy model. This model consists of assigning to each possible instruction, in a given abstraction level (i.e. assembly, bytecode, language, etc.), a consumption value that best summarized how much energy is that particular instruction expected to consume. The consumption values were obtained through external power monitors. This was the strategy followed, for instance, by **vLens** [Li et al., 2013], **eLens** [Hao et al., 2013], and **eCalc** [Hao et al., 2012]. These works proposed an approach where energy models for Java instructions (vLens and eLens) or bytecode instructions (eCalc) are combined with program analysis techniques, in order to estimate energy consumption of Android applications, in a particular

[2]For instance, PowerTutor reports 1 estimation per second only.

and controlled execution scenario.

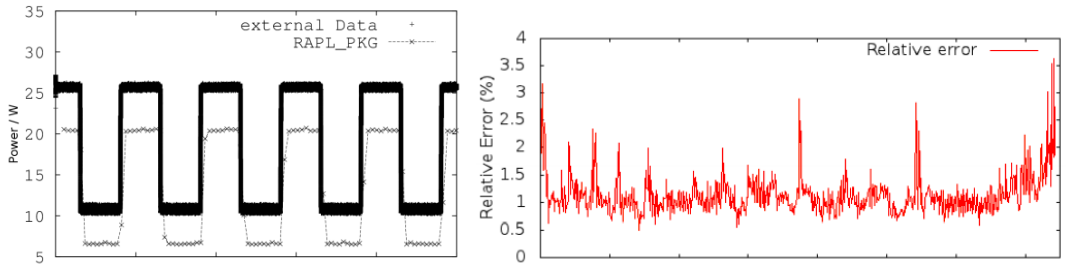
Researchers also explored a data mining approach to create energy profilers, specifically for the Android ecosystem. The idea was to create a technique capable of using test results of several application, and the respective execution trace, to predict the energy consumed by repeated code fragments. This concept was explored with **GreenScaler** [Chowdhury et al., 2018]. This profiler benefits from a large data collection of energy test results, obtained from previous works [Hindle et al., 2014, Chowdhury and Hindle, 2016], and uses data mining to create a model with predictions for several system calls. Upon inspecting which system calls are used when testing an application, GreenScaler can accurately report the expected energy consumed.

These energy models can be useful both for dynamic and static analysis approaches. Nevertheless, they need to be created for every testing architecture (i.e., each smartphone or computer), which can be a costly an undesired task. This raised a demand for generic energy profilers, which could provide accurate estimations for several testing architectures, without the need to create models beforehand. In order to do this, a deep understanding of the hardware's energy behavior is necessary, hence hardware manufactures were the ones capable of developing these type of profilers.

Intel and Qualcomm each developed their own profilers, **RAPL** (**R**unning **A**verage **P**ower **L**imit) [Pandruvada, 2014, David et al., 2010] and **Trepn** [3], respectively. The first one can be used in desktops running with Intel processors, and reports accurate estimations for CPU and DRAM consumption; the second is suited for Android devices that use Snapdragon CPUs, and its estimations are obtained through a PMIC (Power Management Integrated Circuit), hence it consider the whole consumption of the device. RAPL is capable of collecting around 100 estimations per second (check with Rui), while Trepn collects a new estimation every 100 milliseconds. The values reported by RAPL can be obtained through a C-based interface, while it has also been adapted to support interfaces for other languages, such as Java [Liu et al., 2015] and Haskell [Lima et al., 2016]. The accuracy of the reported estimations was already validated for both profilers in several studies [Hähnel et al., 2012, Desrochers et al., 2016, Hoque et al., 2015, Fahad et al., 2019]. A summary of these studies is displayed in Figure 2.2.

The concept of hardware manufactures providing information regarding the energy consump-

[3] Trepn Power Profiler's website: https://developer.qualcomm.com/trepn-profiler (visited on October 19, 2020).



(a) RAPL accuracy, compared with external measurements.
(obtained from [Hähnel et al., 2012])

Profiler	Disp	CPU	GPU	Reported Accuracy
Trepn	✓	✓	✓	99%
V-edge	✓	✓	✗	86%
BatteryStats	✓	✓	✗	Not Reported
PowerBooter	✓	✓	✗	96%
PowerTutor	✓	✓	✗	97.5%
DevScope	✓	✓	✗	95%
AppScope	✓	✓	✗	92%

Brightness	Trepn Profiler	Monsoon	Difference
20%	1245 mW	1247 mW	-0.2%
30%	1321 mW	1340 mW	-1.4%
40%	1408 mW	1490 mW	-5.5%
50%	1517 mW	1550 mW	-2.1%
60%	1627 mW	1643 mW	-1.0%
70%	1729 mW	1773 mW	-2.5%
80%	1838 mW	1857 mW	-1.0%
90%	1951 mW	2002 mW	-2.5%
100%	2088 mW	2148 mW	-2.8%

(b) Trepn accuracy, compared with external measurements.
(obtained from [Hoque et al., 2015])

Figure 2.2: Reported Accuracy for RAPL and Trepn

tion of their devices is even foreseen in Android by design. In its open source project documentation, it is stated that "device manufacturers must provide a component power profile"[4] in a specific location in the Android filesystem. In that file, the current required by different components (such as CPU or WiFi) should be specified. This power profile is used, for instance, by **PETrA** [Di Nucci et al., 2017b, Di Nucci et al., 2017a]. This tool provides mechanisms to automatically test Android applications and report the energy consumed by each method (on average), and its built-in energy profiler uses these power profiles as an energy model. Its major downside is the fact that the power profiles are not always provided by manufactures, and there is no provided knowledge regarding the accuracy of the existing ones.

Finally, recent versions of Android Studio, the IDE used for developing Android applications, includes an energy profiler, called **Battery Historian**[5]. This profiler was one of the results from project Volta, which aimed at improving energy efficiency of Android applications. Its sampling rate is not as high as other profilers, which makes it not the best choice for analyzing the energy efficiency of code blocks. Nevertheless, it provides several additional details on what is causing battery drains in an application, as depicted in Figure 2.3.

[4] Android Power Profile: https://source.android.com/devices/tech/power/values (visited on October 19, 2020).
[5] BatteryHistorian webpage: https://developer.android.com/topic/performance/power/battery-historian (visited on October 19, 2020).

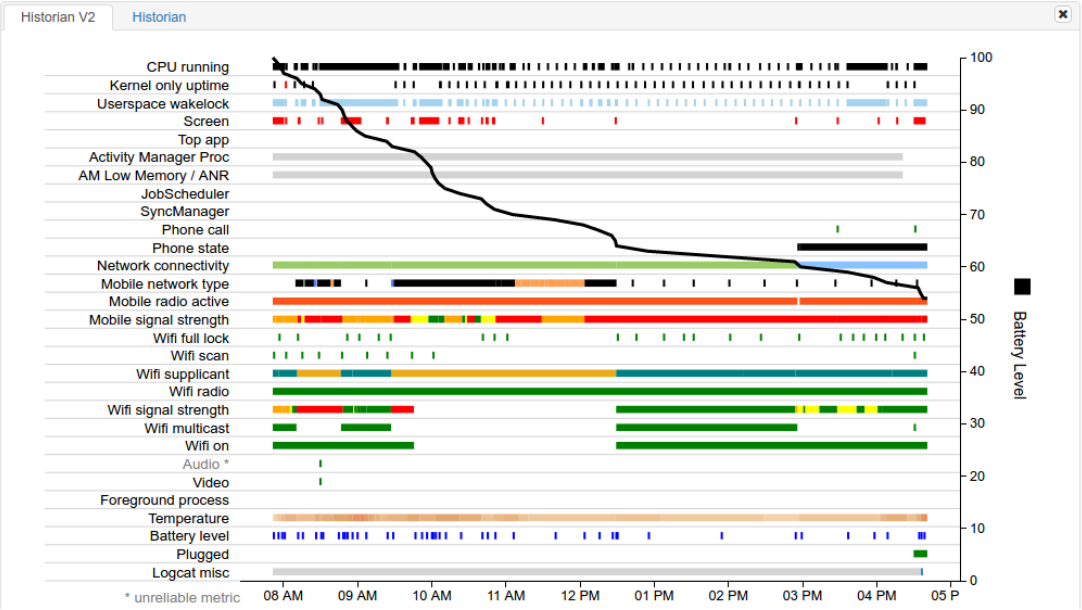


Figure 2.3: The Output from Battery Historian

The appearance of the most varied types of profilers is a result of the growing interest in studying how energy and software are related. This was specially evident in the Android environment, which is the most widely used mobile operating system[6]. Deciding what monitor/profiler to use in a research work depends on its main goals. We explained the advantages of each one, considering the accuracy and granularity level of the results, the cost of creating an experimental setup, the easiness of replicating a study, and the differentiating features each energy monitoring tool provides.

2.2 Detection of Energy-Greedy Code

Profiling energy consumption is the first step towards understanding how to reduce it. The idea is to examine how energy consumption varies during the time it takes for software to run, while keeping track of what code blocks (i.e., methods, procedures, API calls, etc.) were executed in each timespan. This allows both researchers and developers to understand the energy behavior of their code.

Nevertheless, some key concerns must be considered when profiling energy:

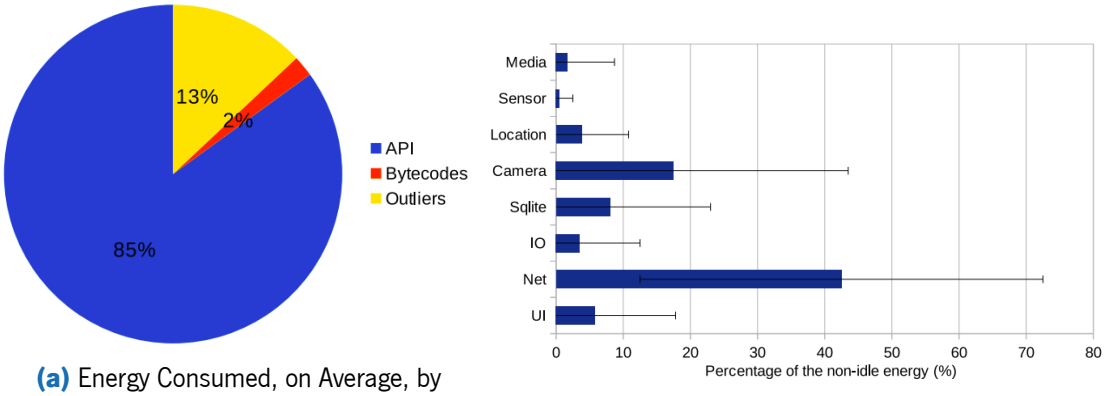
- **The quality and diversity of the tests**: the same code block can behave differently

[6]Mobile OS market shares: https://gs.statcounter.com/os-market-share/mobile/worldwide (visited on October 19, 2020).

in different executions; for instance, the effort employed by a method responsible for encrypting a message varies depending on several factors, such as the cipher used, the size of the encryption key, or the message itself.

- **High consumptions can also be necessary**: if a given code block consumes far more than others, it does not mean that it is excessive; there might not be any room for improvement.
- **Comparisons should be context-aware**: when comparing the consumption of code blocks, it is important not to mix contexts (e.g., compare an external API call for file compression with an arithmetic expression).
- **Focus on most used code blocks**: if a code block tends to consume less energy than others, but is extensively used in all execution scenarios, it also means that any minor improvement can result in a big impact on overall consumption.

Several works explored this energy profiling strategy/concept, in order to identify actual energy-greedy components in software (often called **energy bugs/leaks** or **energy hotspots**). For instance, in [Li et al., 2014a] a large-scale study was conducted, with different usage scenarios being executed on 405 Android applications. This study aimed at understanding, among other things, how much energy is consumed in the idle state, how much of non-idle energy is consumed by developer-written code or system APIs, and how these consumptions are divided among different components. They concluded that, on average, 61% of the energy in consumed in idle states, system APIs dominate non-idle consumption, and the network is the most energy consuming component. An overview of their results is depicted in Figure 2.5.



(a) Energy Consumed, on Average, by Different Types of Code.

(b) Average Energy Consumption, Divided by Components.

Figure 2.4: How Energy is Consumed Inside Real-World Android Apps (obtained from [Li et al., 2014a])

Other studies, similar to the previous mentioned one, were carried out to further understand how energy behaves in Android, and therefore detect potential energy inefficiencies. In [Gui et al., 2015], 21 real world applications were extensively tested, to assess the energy impact of ad-related code; each application was compared with its ad-free version (i.e., after removing ad-related code), and the observations (summarized in Figure 2.5) allowed authors to conclude that including ads can substantially increase the amount of dissipated energy. Likewise, another study [Linares-Vásquez et al., 2014] aimed at classifying Android APIs, based on the amount of energy they consume and the context on which they are used; the authors further examined the results, through data mining mechanisms, and were able to identify a few usage scenarios for the API calls where the anomalous consumption could be avoided.

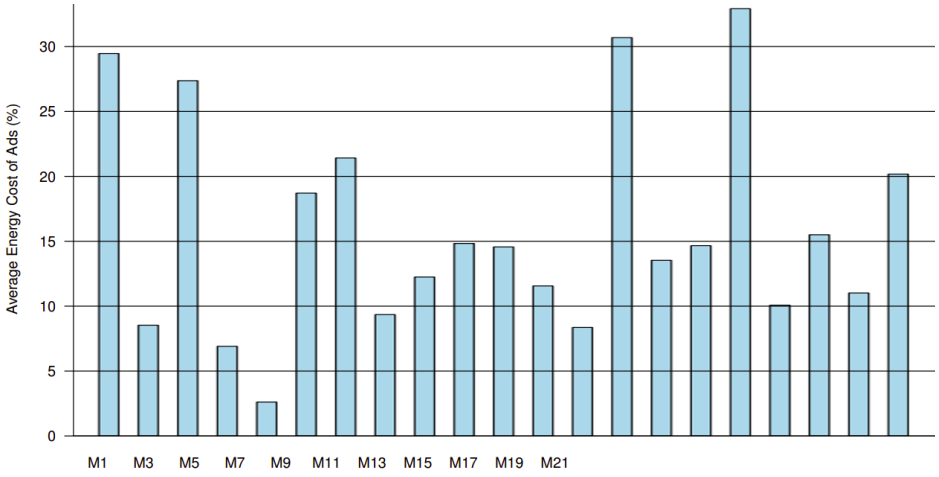


Figure 2.5: The Energy Cost on Including Ads in Android Apps (obtained from [Gui et al., 2015])

The aforementioned works provide high-level observations and insights, that can be exploited by developers to improve the energy efficiency of their code. Nevertheless, it is known that developers also seek for tools that can help them examine specific aspects of their code related to energy issues [Manotas et al., 2016, Pinto et al., 2014, Pinto and Castor, 2017]. As such, researchers looked for ways to develop these multi-purpose energy bug detectors.

SPELL [Pereira, 2018, Pereira, 2018] is described as "an energy leak localization technique", which considers the program spectrum, obtained at the testing phase, to identify potential energy inefficient code. In each test run, SPELL considers, for each tracked code block, the amount of energy it consumed, its execution time, and the number of times is was executed. It then combines this information to estimate the likelihood of each block being responsible for increasing the energy overall energy consumption. Currently, it is developed for examining meth-

ods in Java desktop applications. An empirical study was employed to assess its accuracy, and the usefulness to developers, both with favorable results.

In the Android scope, the framework developed in [Banerjee et al., 2014] is capable of automatically test an application, with several different scenarios, and pinpoint in the code two types of energy inefficiencies: (i) hotspots, which are described as situation where the energy is abnormally high when hardware resources have little usage, and (ii) wakelocks, which prevents an application without user activity to became idle and therefore save energy.

Constructing large datasets of energy related data, that can be mined in order to detect energy hotspots in software, is also an employed strategy, particularly for Android. **GreenSource** [Rua et al., 2019b] was created to encompass a large collection of data, obtained from automatically running several tests on several Android applications. For each test, different metrics are calculated, such as energy consumed, method calls, hardware components' usage, among several others. Similarly, **GreenHub** [Matalonga et al., 2019] also contains a large number of Android-related data, targeted for energy analysis through data mining. The difference is that its data was obtained from real-world usage of Android phones, and therefore covers a wider variety of usage contexts, while contains only application and system level information.

Both GreenSource and GreenHub are ongoing projects. At the time, only the data repositories are available, and no findings were provided resulting from any analysis with their data. Nevertheless, since both are, to the best of our knowledge, the first repositories of their kind, it is expected that novel findings can arise soon from examining their data.

2.3 Adjusting Software and Improving Energy

As explained in the previous section, the ultimate purpose of profiling energy is to locate potential energy optimization chances in the code. The following step is to further inspect the code, and reason about alternatives that can make it more energy efficient. Hence, researchers developed several studies where they show that the energy consumption of a software system can be significantly influenced by a lot of code-related factors, while providing insights on less energy-greedy alternatives.

Deciding on what programming language to use can result in energy savings in the long term,

as it was already proven that energy consumption varies a lot between languages, even when implementing the exact same algorithms [Couto et al., 2017b, Pereira et al., 2017b]. A summary of this study is included in Table 2.1. A total of 13 benchmark problems were implemented in 27 languages, and executed while collecting data regarding energy consumption, execution time, and memory usage. At the end, languages were ranked according to how good their results were, when compared with other languages.

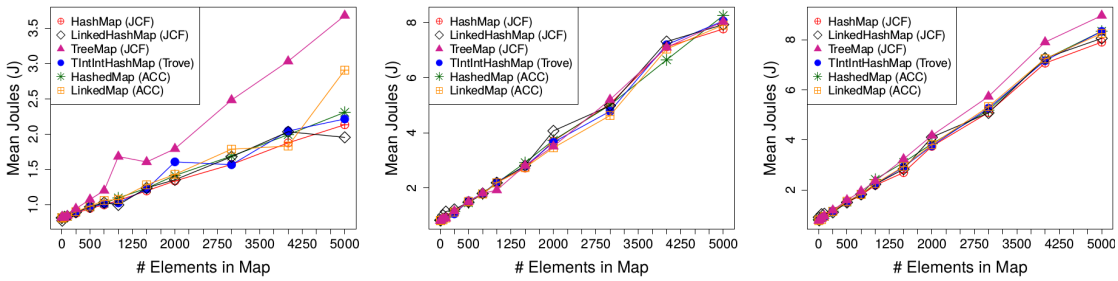
Table 2.1: Language Rankings for Different Optimization Goals (obtained from [Pereira et al., 2017b]).

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		

Another factor that can highly influence energy is the choosing between data structures. A few studies were conducted for the Java programming language, where it was analyzed the energy impact of choosing different Maps, Lists, or Sets from the Java Collection Framework (JCF). In [Hasan et al., 2016], for a subset of collections from the JCF, the consumption of insertions, iterations, and lookups was analyzed, when executed on objects with a variable number of elements. To exemplify the results, Figure 2.6 shows the ones obtained for different Map and Set collections.

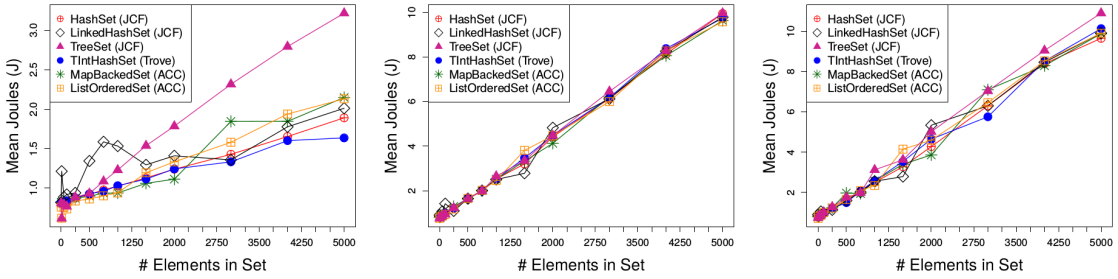
Similar approaches were followed to assess the energy efficiency of Java collections. For instance, the work presented in [Pereira et al., 2016] complements the previous referred one, in the sense that all collections from the JCF were analyzed, and all methods provided by each collection were analyzed[7]; in [Pinto and Castor, 2014], Java collections from the JCF described as thread-safe were tested in a multi-core execution environment, to assess the trade-offs between performance and energy efficiency; finally, in [Manotas et al., 2014], the results of comparing

[7]The results of this study are publicly available in: https://greenlab.di.uminho.pt/collections/ (visited on October 19, 2020).



(a) Insertion (b) Iteration (c) Random Query

(a) Insertions, Iterations, and Lookups on Maps.



(a) Insertion (b) Iteration (c) Random Query

(b) Insertions, Iterations, and Lookups on Sets.

Figure 2.6: Energy Consumed by Standard Operations on JCF collections (obtained from [Hasan et al., 2016])

the energy efficiency of alternative Java collections are used to create a decision support framework, which can automatically optimizes Java code to use the most energy-efficient collections, according to the context on which they are used.

The energy efficiency of data structure was not only a concern for Java collections. In fact, similar studies were performed for Haskell data collections [Melfe et al., 2018, Melfe et al., 2018]. Following a previous work, where the energy efficiency of Haskell programs was analyzed from a strictness and concurrency perspectives [Lima et al., 2016], these works evaluate if strictness is a better approach than laziness when running standard operations over data structure. After running an empirical experiment over standard data structure libraries for Haskell, the authors concluded that "strict evaluation for most tasks as the default approach, specially when iterating over elements in *Maps*". In Figure 2.7 contains a subset of the results obtained, where it is clear that using strict evaluation enhances energy efficiency on most operations from a Map structure.

As pointed out several times already, energy-related research tends to be more targeted for mobile devices, such as Android-based ones. Reducing energy directly converts into increasing battery uptime, which is critical in this context. We have already explored several works that aid developers in increasing the energy efficiency of their applications. Nevertheless, understanding

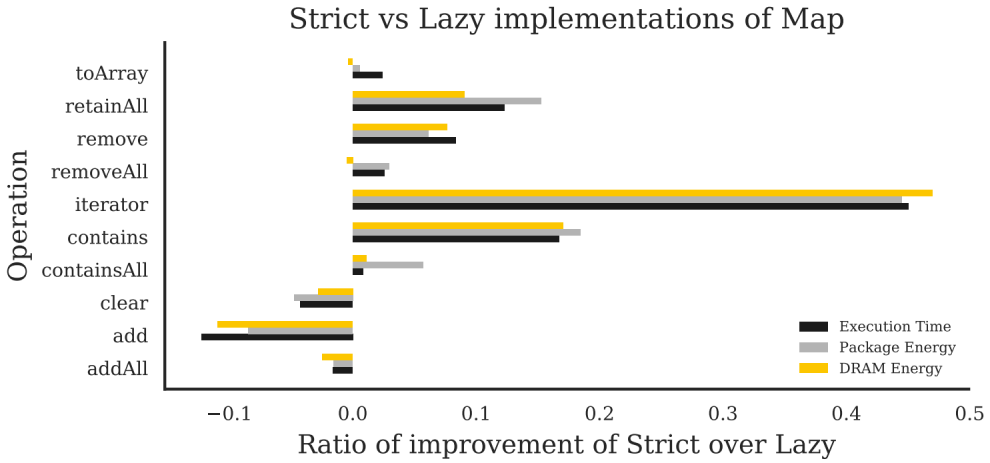


Figure 2.7: Energy Impact of Strict vs Lazy Evaluation on Haskell Data Structures (obtained from [Melfe et al., 2018])

how different programming practices impact on the energy consumption of Android applications is likely to be the most explored subject in this area over the last years. Hence, an energy-greedy programming practice is defined as a coding practice for which an alternative exists, and was proved to be more energy efficient.

Ultimately, two types of energy-greedy programming practices have been studied in Android:

- **Code Patterns**: code-level patterns, usually associated to specific functionalities, and where applying an alternative can be achieved by changing very small portions of code; these are more widely explored patterns, in several different works [Vekris et al., 2012, Li and Halfond, 2014, Carette et al., 2017, Cruz and Abreu, 2017, Jiang et al., 2017, Morales et al., 2018, Saborido et al., 2018, Cruz et al., 2019, Palomba et al., 2019, Couto et al., 2020a].
- **Design Patterns**: high-level patterns, mostly related to the program structure, and where the suggested alternative involves changing the semantics of the program/application; fewer works [Morales et al., 2018, Cruz and Abreu, 2019a] explored their energy impact.

In both types of patterns, the authors that studied their energy impact always provide a less energy-greedy alternative. Hence, developers can look at this collection of research works to understand how to improve their code, in order to reduce the overall energy consumption of their applications. For **code patterns**, a few tools have been developed which are capable of automatic replace a patterns for a known less-greedy alternative [Cruz et al., 2017, Cruz and Abreu, 2018, Couto et al., 2020a].

Although mobile devices is the most explored subject when studying energy code patterns, the fact is that this concern has also been been extended to other scopes, particularly to desktop applications [Sahin et al., 2012, Feitosa et al., 2017]. In [Sahin et al., 2012], an initial effort was performed in order to understand the energy impact of design patterns. A prototype tool was developed, which could profile energy consumption of software, comparing the impact of an approach using design patterns against not using design patterns.

Gang of Four (GoF) design patterns [Gamma et al., 1994] and their relation with energy consumption was also studied. The work presented in [Feitosa et al., 2017] focused on comparing the energy consumed by 169 methods involved in 2 GoF patterns against a version of the same methods that do not leverage the use of GoF patterns. Their findings indicate that using these 2 GoF patterns tends to increase energy consumed.

2.4 Energy Behavior Prediction

There is a variety of scenarios where measuring or profiling energy consumption in software is an arduous and undesirable task, and sometimes even infeasible. This happens when it takes too much time to adequately perform the analysis, or if there is no guarantee that the analysis will cover all the intended use cases. An example of such a scenario is when the goal is to deeply assess if executing a particular usage scenario on an application will always consume less than a certain amount of energy. This requires a lot of time to carefully design and run the experiment, and the results might not reflect the entirety of possibilities.

Aware of these challenged, researchers worked on different strategies to address them. In Android applications, for instance, a couple of works have focused on building mechanisms capable of accurately estimate the amount of energy required by a specific execution path [Hao et al., 2012, Hao et al., 2013]. This was achieved by building reliable energy models, where all possible instructions have an associated energy consumption value. Hence, upon inspecting the intended execution path, using traditional program analysis mechanisms, it is possible to obtain a accurate estimation for how much energy will be required. The difference between these 2 works is the abstraction level of the instruction: the first one examines bytecode instruction [Hao et al., 2012], while the second considers Java instructions [Hao et al., 2013].

Other approaches were followed intended for predicting energy behavior in Android applica-

tions. The work developed by Nakajima [Nakajima, 2013, Nakajima, 2015a, Nakajima, 2015b] represents a strategy to analyze energy consumption as a model-checking problem, showing that it could be possible to provide a consumption bound for a specific execution scenario, by defining the energy behavior of different hardware components as an automaton. The presented concepts in this works have not been, to the best of our knowledge, subject to any experimental validation. Hence, it is not possible yet to assess the accuracy and applicability of the techniques.

Understanding energy consumption is not only critical for Android devices and applications. Embedded devices also have battery uptime as their major limitation, particularly the ones used in complex IoT systems and infrastructures, where the energy storing capacity is much more reduced. Accurately predicting energy consumption is a key concern for these devices, since it is not feasible to measure it at runtime, and with an accurate estimation it is possible to determine the amount of work they are expected to accomplish (e.g., how many samples can be collected until the device runs out of battery).

In the scope of energy harvesting devices (i.e., devices that can collect and store themselves the energy required to operate), which are particular type of IoT devices, a considerable effort has been made towards modeling their behavior to predict energy consumption. For instance, CleanCut [Colin and Lucia, 2018] analyzes a program suited to execute in these devices, and is capable of determining the probability of that program failing to execute entirely, through combining static program analysis techniques with a instruction-oriented energy model. It also suggests how to split the program in independent tasks, using a previously developed approach called Chain [Colin and Lucia, 2016], so that if the device runs out of energy it can resume by repeating the interrupted task, without compromising the program's semantics. This suggestion also results in energy savings, as it removes the need for repeating the whole program.

Our work described in Chapter 3 [Couto et al., 2017a, Couto et al., 2020b], where we describe a technique capable of predicting the energy consumption in software families in a worst-case scenario, is based partially on the aforementioned works. For instance, the energy model creation follows the same approach as in [Hao et al., 2013], although it is suited for instructions in the C language. Moreover, our technique provides an upper-bounded energy estimation for each analyzed program, similarly to what is done in [Colin and Lucia, 2018]. Nevertheless, the execution path for the worst-case scenario is inferred, and does not need to be provided.

2.5 Additional Energy Optimization Approaches

The summary of research works presented in the previous sections covers what we argue as being the 4 main goals in this research area: i) measuring/profiling energy, ii) reasoning about consumption to detect inefficient code, iii) prediction/estimating energy and when is it appropriate to do it, iv) and providing information to developers on how to improve their code and reduce their energy footprint. Nevertheless, in respect to the last goal, there are other research works which focused on developing more elaborated energy optimization approaches for developers to follow.

Bundling computational intensive tasks has been the subject of a couple of research works. In [Li et al., 2016], it was suggested that bundling HTTP requests in Android applications would result in energy savings. The rationale behind this is the fact that, with request bundling, the initial and final efforts required to (respectively) open and close the connection would be reduced, since it would not be done once per request. Additionally, by performing the requests all at once the caching advantages would be maximized. An overview of the obtained results is depicted in Figure 2.8.

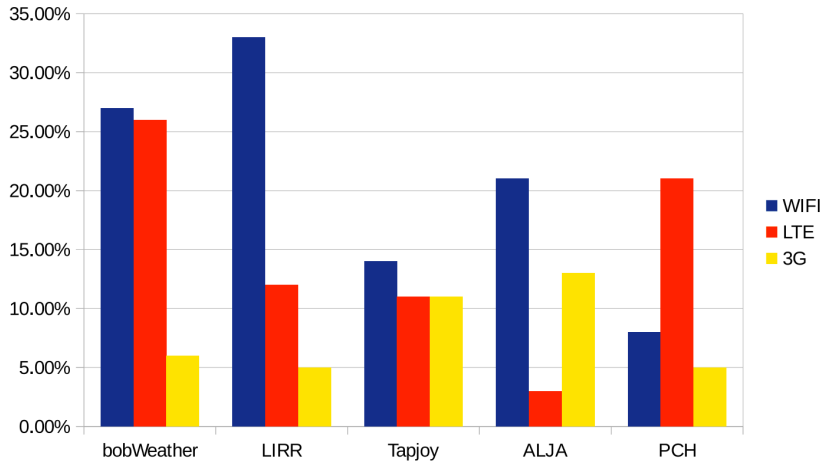


Figure 2.8: Energy Savings (in %) Obtained From Bundling HTTP Requests. (obtained from [Li et al., 2016])

The bundling strategy was again explored in the Android scope, but applied to view update events [Chowdhury et al., 2019]. In other words, the traditional Model-View-Controller (MVC) pattern was transformed in a Model-View-Presenter (MVP) pattern: each View component, instead of directly obtaining the data required to render from the Model component, would request it to the Presenter component. Ultimately, this means that the Presenter can cache already retrieved

data, **bundle** requests targeted to the same Model entity, and even **drop** (i.e., filter) redundant requests[8]. The overall results of this approach tested in 4 applications, which are included in Figure 2.9, suggest that there is a tendency for obtaining energy savings after some time when implementing bundling/dropping strategies.

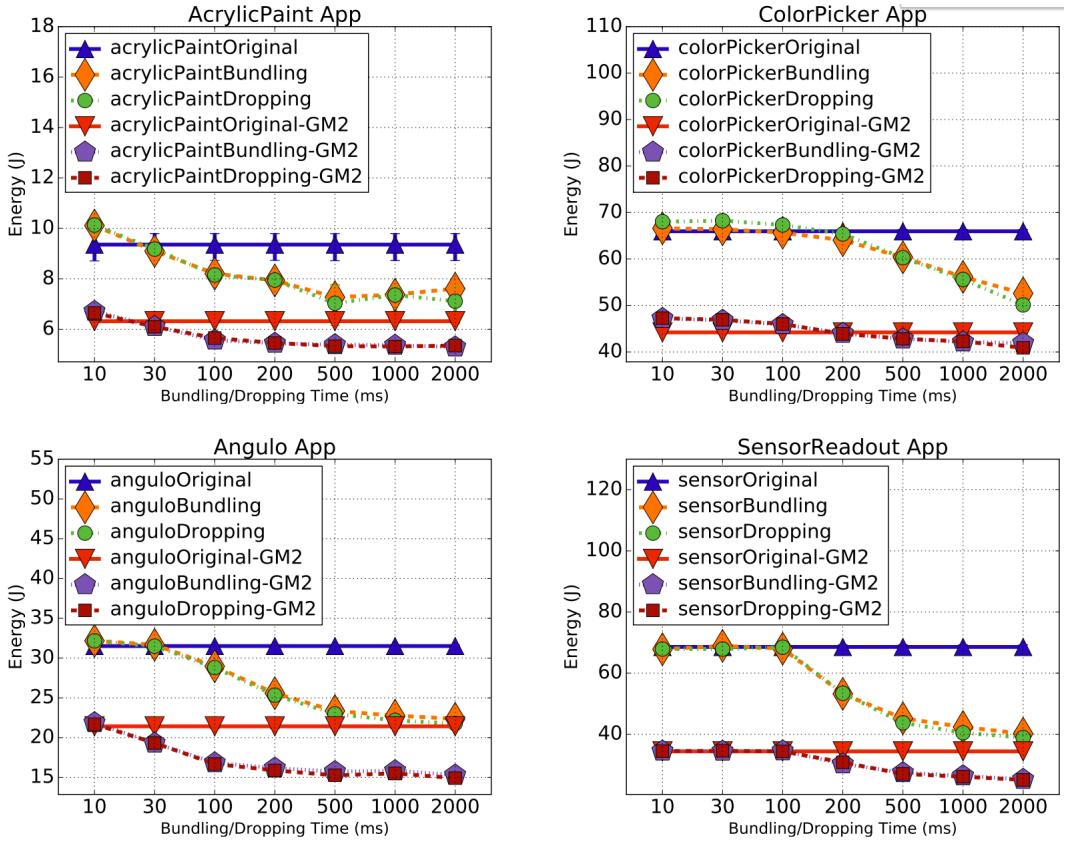


Figure 2.9: Energy Impact of Bundling/Dropping Events in 4 Android Applications. (obtained from [Chowdhury et al., 2019])

Memoization was also explored in another study [Rua et al., 2019a], to assess if it could be used as an energy optimization technique. In this preliminary study, 3 Android applications where the subject to an experiment. A few of their methods, identified as having a pure function behavior[9], were executed several times in two ways: with and without using a built-in memoization mechanism. The energy consumption for both executions of each method was compared, and the results indicated that memoization did help reduce energy in most cases. Although this was only a preliminary and simplistic experiment, and the obtained conclusions might not be generally applicable, it clearly showed the potential of that memoization as an energy optimization approach,

[8] For instance, if the Presenter knows that a certain request would not retrieve updated data.

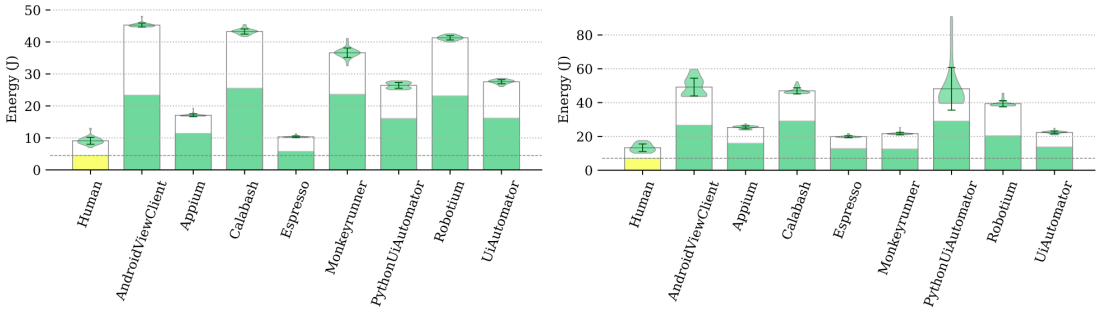
[9] I.e., the return value is the same for the same arguments, and does not perform mutations on any existing variable.

and there might be useful to further exploring it.

Implementing the aforementioned 3 approaches is obviously not a straightforward task. There is also an associated side effect with the first 2, which is the fact that the behavior of the Android applications using them would be affected. Hence, developers need to be a bit more cautious when implementing them, which is why we chose to separate these improvement approaches from the ones presented in Section 2.3.

Another aspect that has caught the attention of researchers is the amount of energy consumed in the testing phase. The importance of building a thorough and reliable collection of tests, which can ideally cover every aspect of a software system, is acknowledged by developers. Nevertheless, the extensiveness and complexity of these tests makes the testing phase an intensive task for any device running them. As such, researchers studied how to optimize this task, without compromising its integrity.

An initial effort was made for in situ testing for embedded devices [Li et al., 2014b]. As embedded devices are known for having a very limited battery power, and this particular testing approach is executed on running devices, the study proposes a test suite minimization technique to reduce the energy consumption to the minimum possible, while maintain the code coverage. A similar approach based on test minimization was also followed in [Jabbarvand et al., 2016], although the target were Android applications and the optimization was intended not for the target device, but for the development equipment.



(a) Energy Consumption of swipe.

(b) Energy Consumption of long tap.

Figure 2.10: Energy Footprint of UI Testing Frameworks: swipe and long tap tasks (obtained from [Cruz and Abreu, 2019b])

Finally, still in the Android scope, the energy impact of choosing between different testing frameworks was also addressed [Cruz and Abreu, 2019b]. In this study, 8 commonly used UI automation testing frameworks for Android were compared, and the impact on choosing between

one of them is quite significant: the energy consumption increase can be up to 2200%. The authors concluded that the framework Espresso is the most energy efficient, despite having some limitations on the possible interactions, and when considering a full set of UI testing features, Appium, Monkeyrunner, or UIAutomator are the best energy-efficient choices. Figure 2.10 shows a comparison of the energy efficiency of all 8 frameworks, when performing 2 UI tasks: swipe and long tap.

3 Energy Estimations in Software Product Lines

Optimizing software to become (more) energy efficient is an important concern for software industry. Although several techniques have been proposed to measure energy consumption within software engineering, little work has specifically addressed Software Product Lines (SPLs). SPLs are a widely used software development approach, where the core concept is to study the systematic development of products that can be deployed in a variable way, e.g., to include different features for different clients. The traditional approach to measure energy consumption in SPLs is to generate and individually measure all products, which given their large number is impractical.

We present a technique, implemented in a tool, to statically estimate the worst-case energy consumption for SPLs. The goal is to reason about energy consumption in all products of a SPL, without having to individually analyze each product. Our technique combines static analysis and worst-case prediction with energy consumption analysis, to analyze products in a feature-sensitive manner: a feature used in several products is analyzed only once, while the energy consumption is estimated once per product.

On a detailed evaluation using 4 real-world SPLs, and considering a total of 99 products, our results show that the tool was able to estimate the worst-case energy consumption with a mean error percentage of 17.3% and a standard deviation of 11.2%.

3.1 Introduction

The worldwide energy provision model is currently supported by unsustainable sources. The evidence provided by recent reports [Ene, 2019] is that 81% of the energy consumed in 2018 has been obtained from nonrenewable sources, while electricity consumption has essentially doubled in the last 30 years. In addition, with the growing investment interest on ICT infrastructures and applications, it does not come as surprise that this sector, which is nowadays responsible for between 5% and 9% of the worldwide energy consumption, could increase this number up to 20% of the world's produced energy in the next decade [ICT, 2018].

Energy is the limiting resource in a wide range of computing systems, from embedded sensors, to mobile phones, IoT devices or even data centers. Considering global data centers alone, these have used roughly 416 tera-watts in 2016 (4.16×10^{14} watts, or about 3% of the total produced electricity) [Rda, 2017].

In addition, the software components supporting modern computing systems are getting more and more complex, supplying more and more business areas and running on a wider variety of devices and system architectures. One example is found in Android-based mobile devices, with the same base system being deployed in multiples devices, yet adapted to better fit the devices needs and characteristics. To reduce the effort in creating multiple software versions, it is important that shared components and assets are reused in a controlled and systematic manner.

Software Product Lines (SPL) have emerged as an important software engineering discipline allowing the development of software that shares a common set of features. Thus, SPLs are particularly suitable to develop software where individual products target specific software features and computing architectures/devices, while sharing common characteristics (i.e., core features). Through the exploration of Dynamic Software Product Lines (DSPLs), this approach was adapted in order to comprise mechanisms capable of managing variability at the runtime phase, in a manual or automatic manner [Capilla et al., 2014].

In practice, SPLs are widely used to create operational flight programs [Sharp, 1998], software to control gasoline system engines [Steger et al., 2004], medical systems for image-supported diagnosis [Pronk, 1999], or even television sets' operating systems [van Ommering, 2002], among

several other examples[1].

Although several techniques have been proposed to measure, reason about and improve the energy efficiency within software and language engineering [Pereira et al., 2017a, Jabbarvand et al., 2015, Li et al., 2014b, Pereira et al., 2017b, Hu et al., 2018], only recently there has been an effort towards addressing this issue in SPLs [Couto et al., 2017a, Horcas et al., 2019]. In a nutshell, a SPL development approach was presented in [Horcas et al., 2019], mainly focused on building a DSPL that can adapt itself with energy-saving capacities, through establishing a relation between variability features and their energy impact and continuous monitoring the context on which the SPL is used. Regarding energy analysis of all products within a SPL, the first emerging work [Couto et al., 2017a] attempted to explore static analysis to reason about such products, in order to obtain for each one accurate energy consumption estimations.

Using static analysis is a clear improvement over a brute-force approach, which would consist of generating all products within a SPL, and use common techniques to measure the energy consumption of each product, even if it shares features with other products. Indeed, this approach is in practice too costly, since, for each product, it is necessary to assess it at runtime, which can become unfeasible depending on the size and complexity of the SPL.

When analyzing highly complex SPLs, an approach frequently adopted by the community is to use sampling techniques [Varshosaz et al., 2018]: only part of the existing products are built and tested, assuring that all features are properly exercised. This approach, however, is not recommended when analyzing non-functional properties of SPLs, such as energy consumption, due to the high impact that feature interactions have on such properties [Sarkar et al., 2015, Siegmund et al., 2015].

In this chapter, we further explore static analysis techniques to reason about energy efficiency in the context of SPLs. Our approach targets SPLs that are developed in C, using conditional compilation to express variability.

Our goal is to analyze SPLs in a feature-sensitive manner, i.e., the code related to a particular feature is analyzed only once, while the analysis result is reflected for every products where such feature is included. In order to do so, we use SPL static analysis techniques [Brabrand et al., 2012] that work on a control-flow graph representing a set of programs (the SPL), and not just

[1] More well-known examples can be found in the SPLC Hall of Fame: https://splc.net/fame.html (visited on October 19, 2020).

one. This analysis is used to compute 2 things from the source code: i) energy related properties, such as hardware components usage information, and ii) dataflow information, such as loop upper bounds.

The computed properties are then used as input to a Worst-Case Execution Time (WCET) prediction technique, where instead of execution time we compute the energy consumption per product. This is the only step where the analysis is product-oriented. We use a constraint solver to estimate the consumption, where, for each product, we generate a set of constraints. We call this technique the Worst-Case Energy Consumption (WCEC).

Our WCEC analysis technique for products in a SPL implies the existence of an energy consumption model. This model abstractly describes the hardware where the products will run, and the amount of energy needed by different types of instructions to execute. Using such model along with the previously computed properties, our technique computes the energy consumption profile for each product and feature. Thus, it allows the generation of the most energy efficient product that includes/excludes a set of given features.

We performed an evaluation for our technique using 4 real-world SPLs, which allowed us to study in more detail the general accuracy of our approach; this is described in detail in Section 3.5. Upon doing so, we found that the accuracy of our energy model could be improved in certain cases. We adapted a well-known optimization algorithm to find the best way to improve the model, which is explained in detail in Section 3.5.3.

The final results of our assessment show that the energy predicted by our technique and tool for each product is always, as expected, an overestimation of the real consumption, diverging on average by 17.3% with a standard deviation of 11.2%.

The remaining of this chapter is structured as follows: Section 3.2 provides an introduction to static analysis in the context of SPLs. Section 3.3 presents in detail the worst-case energy consumption analysis approach that we propose. Section 3.4 briefly describes the concepts and architecture of the WCEC prototype that we developed. In Section 3.5 we show the results of using the prototype to estimate the WCEC of real SPLs. In Section 3.6, we discuss the threats to the validity of our technique and experiments. Finally, we include the conclusion of our work in Section 3.7.

3.2 Static Analysis in SPLs

The goal of our work is to develop a methodology for statically predicting the energy consumption of products in a SPL. In order to better explain our approach, we first need to explain some introductory concepts that we based our approach on.

In this section, we will explain how static analysis can be achieved in SPLs. We start by explaining SPLs concepts in Section 3.2.1. Then, we will explain the static program analysis concepts in Section 3.2.2, and finally we will present and explain the chosen technique for integrating all the previously stated concepts in order to achieve static analysis in SPLs in Section 3.2.3. The concepts included in this section are mostly inspired from the contents of [Brabrand et al., 2012].

3.2.1 Basic Concepts of Software Product Lines

Research on Software Product Lines has been focused on software engineering methods and techniques designed to manage variability in a software system, in a normalized manner [Brabrand et al., 2012, Bodden et al., 2013]. Using SPLs allows the creation a collection of different software solutions from reusable assets in the same system (i.e., code fragments, visual assets, etc.), using the same means of production.

In order to illustrate the essential aspects of SPLs, let us consider the purchase of a car. In such a situation, once the particular car model is choosen, it is often required to configure it according to preferences and budget. Configuration options are diverse, and often involve choosing the engine type, entertainment system, etc..

Imagine that the following (simplified) configurations are possible for a given car: turbo engines are not available with the basic version; turbo engine is mandatory when choosing air-conditioning.

In the context of SPLs, the `Basic` version, `Turbo` or air-conditioning (`Air`) are often referred to as features.

Thus, a product (e.g., a concrete car that respects the configurations which are possible) is characterized by the set of features which it includes/implements (e.g., `{Car, Basic, Air, Turbo}`). This set of features is called the product configuration, and can be used to generate

several products of the same kind.

The number of possible product configurations may sometimes be restricted by a so-called feature model. This model is a propositional logic formula, which deals with situations such as exclusive features, and features that may imply the inclusion of other features.

For instance, considering our running example and the feature model $\Psi = \texttt{Car} \wedge (\texttt{Basic} \Leftrightarrow \neg\, \texttt{Turbo}) \wedge (\texttt{Air} \Rightarrow \texttt{Turbo})$, the complete set of product configurations is:

$$\{\{Car, Basic\},\ \{Car, Turbo\},\ \{Car, Air, Turbo\}\}$$

One way to define if a code block belongs to a specific feature is by using conditional compilation [Couto et al., 2011]. This technique is based on associating to the code block a preprocessor instruction, an `#ifdef` Φ, where Φ is a propositional logic formula over feature names:

$$\Phi ::= f \in \mathbb{F} | \neg\Phi | \Phi \wedge \Phi | \Phi \vee \Phi$$

Where f is a feature, drawn from a finite alphabet of feature names $\mathbb{F}$. This allows to indicates which features include that code block, or which ones must exclude it.

3.2.2 Static Dataflow Analysis concepts

In order to briefly review classic static program analysis concepts, we will provide illustrations based on the following code example:

```
1    void m() {
2      int i = 0;  int x = 0;
3      x = input();       //1..100
4      #ifdef (A) x*=2;
5      #ifdef (B) x--;
6      while(i < x){ i++; }
7      }
```

Listing 3.1: Example of a SPL method

In this example, conditional compilation primitives are used to define a SPL where two features, `A` and `B`, are available. If `A` is present in a particular product, the value of `x` doubles after obtaining the input; if `B` is present, the value of `x` is decremented.

Every classic static dataflow analysis consists of three components: a) a control-flow graph,

representing the connection between instructions (and on which the analysis is performed); b) a lattice, representing the values of interest for the analysis; and c) transfer functions, responsible for simulating the execution of the program represented by the control-flow graph.

These components, are the input to the fixed-point computation function. This function is responsible for calculating a lattice value for each node of the control-flow graph, which is the fixed point of the transfer functions at that point.

Control Flow Graph (CFG)

This component is an abstraction of the program given as input. A CFG is a directed graph, with the statements of the program to be analyzed as nodes, and the edges representing the flow of control. An edge can be assigned a boolean expression, which is the condition to be verified in order for the program to follow that flow. Figure 3.1 represents the CFG for the product derived from the example in the Listing 3.1, when both features (`A` and `B`) are included.

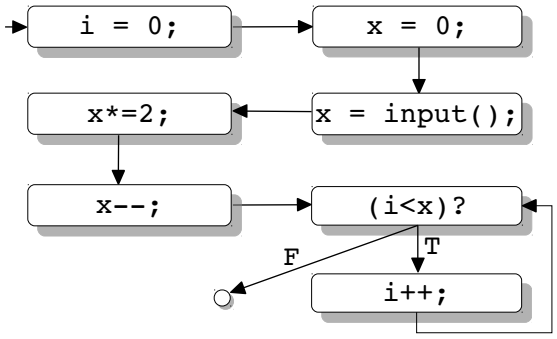


Figure 3.1: Control-Flow Graph for the example in Listing 3.1

Lattice

When performing static dataflow analysis, the calculated information is arranged in a lattice, $L = (D, \sqsubseteq)$, where D is a set of elements and $\sqsubseteq$ is a partial-order on the elements.

Each element represents information relevant for the analysis to be performed. For example, if the goal of the analysis is to check the signal of a variable (called a sign analysis) the element "+" indicates that a value is always positive, while the element "0/+" represents zero or positive. Figure 3.2 represents a lattice for the sign analysis.

There are two special elements in the lattice: $\bot$, at the bottom of the lattice, usually meaning the node was not yet analyzed, and $\top$, at the top of the lattice, usually meaning it is impossible

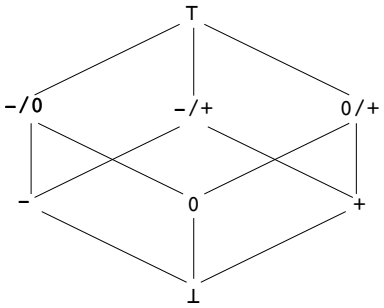


Figure 3.2: Lattice for Sign Analysis

to determine the lattice element for that node.

The partial order induces a least upper bound operator, represented by $\sqcup$. This operator allows information to be combined during the analysis, when a node has more than one entry point (for example when there is a loop or an if). For instance, in sign analysis we have: $\bot \sqcup 0 = 0$, $0 \sqcup + = 0/+$, and $- \sqcup 0/+ = \top$.

Transfer/Update Functions

A transfer function is a monotone function which simulates the execution of a statement (with respect to what is being analyzed). Each statement has an associated transfer function which receives one or more lattice elements and uses it to compute a new element associated to the statement. The input elements for the function are the lattice elements associated to the predecessors or successors relevant for the analysis. For example, if we want to perform the sign analysis for the example in Figure 3.1 the transfer function for the statement `i++` would be:

$$f_{\mathtt{x}--}(l) = \begin{cases} \top & : l \in \{-/+, 0/+, \top\} \\ - & : l \in \{-, -/0, 0\} \\ 0/+ & : l = + \\ \bot & : l = \bot \end{cases}$$

In this case, l is the lattice element of the only predecessor of `x--`. If that element is for example $0/-$, it means that before entering the statement `x--` the value of `x` was either zero or a negative value. Given those possibilities, decrementing `x` will only result in a negative value, as the function shows. In order to assure that the analysis is well-defined, all the transfer functions

must obey the monotony property.

Once every transfer function is defined, the first step of the analysis will be to transform the CFG into a whole-program transfer function, T, which contains the transfer functions for all points of the program to be analyzed. Equation 3.3a shows the whole-program transfer function for sign analysis of the example in Figure 3.1.

The last step of static analysis will then be to calculate the fixed point of the whole-program transfer function. Table 3.3b shows the result of such computation for the example in Figure 3.1, using function T. This analysis enables us to determine the sign of the variable `x` in every statement of the program.

$$T\begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \end{pmatrix} = \begin{pmatrix} f_{\texttt{i=0}}(\bot) \\ f_{\texttt{x=0}}(a) \\ f_{\texttt{x=input}}(b) \\ f_{\texttt{x*=2}}(c) \\ f_{\texttt{x--}}(c) \\ (d \sqcup f) \\ f_{\texttt{i++}}(e) \end{pmatrix}$$

(a) Whole-program Transfer Function

a	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
b	$\bot$	0	0	0	0	0	0
c	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
d	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$
e	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
f	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$
g	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
	T^0	T^1	T^2	T^3	T^4	$T^5 = T^6$	

(b) Fixed-point Computation

Figure 3.3: Whole-program transfer function and sign analysis (as fixed point computation) for the example in Figure 3.1

3.2.3 Static Analysis in Software Product Lines

Static analysis, as explained in Section 3.2.2, can only be used in programs with no included variability. In a SPL, there is a need to make the analysis feature aware, so it can compute the results for all products at once.

As explained in [Brabrand et al., 2012], there are different ways to achieve feature-aware static analysis. For both efficiency and ease of implementation, in our context we have chosen to adopt Simultaneous Feature-Sensitive Analysis. In the following paragraphs, we will explain what this technique consists of and how it combines static dataflow analysis with feature awareness.

Simultaneous Feature-Sensitive Analysis

A SPL can be analyzed by building all possible products and analyzing them individually. However,

this is not desirable as a SPL with a substantial number of features will give rise to a considerable number of products. It is possible to avoid building all produts by making the dataflow analysis feature sensitive. In order for this, there are a few changes that need to be made to the static dataflow analysis components.

The CFG needs to be changed in a way that it is possible to know if a statement is feature dependent or not. In order to do this, each node of the CFG has assigned the set of features which implement the corresponding statement. Each node will now have a pair $(S, \mathscr{F})$, where S (e.g., x*=2) is the statement and $\mathscr{F}$ (e.g., $[\![A]\!]$) is the list of features where S is implemented. Unconditional statements will have the list of all configurations, represented as $[\![\ \mathtt{TRUE}\ ]\!]$.

The lattice will also change, since it is now necessary to know which products are affected by the execution of every instruction. This can be achieved using a so-called lifted-lattice, which will basically maintain for every node one lattice element per valid product configuration. This way it is possible to know the result of the analysis on all products. Similarly, the transfer functions will also be lifted, i.e., they will only be applied to the product configurations which include the statement being analyzed.

Figure 3.4 shows, for a part of the example of Listing 3.1, the final result of the fixed point computation, performed for the sign analysis, using the Simultaneous Feature-Sensitive approach. The result here is expressed as an annotated CFG. At the exit of each node there is a lifted lattice with an element associated to each possible product of the SPL, where each element is the result of the transfer function associated to the node.

3.3 Static Energy Analysis in SPLs

In the last section, we explained how static analysis in SPLs can be performed by combining typical state-of-the-art static analysis techniques with the components that characterize a SPL, such as features and feature models.

However, all the presented concepts and techniques were introduced for the purpose of static analysis: predicting whether a property is true or maintained after a certain instruction is executed. In other words, static analysis, as presented until now, enables the possibility of predicting the behavior of all products in a SPL by analyzing their code.

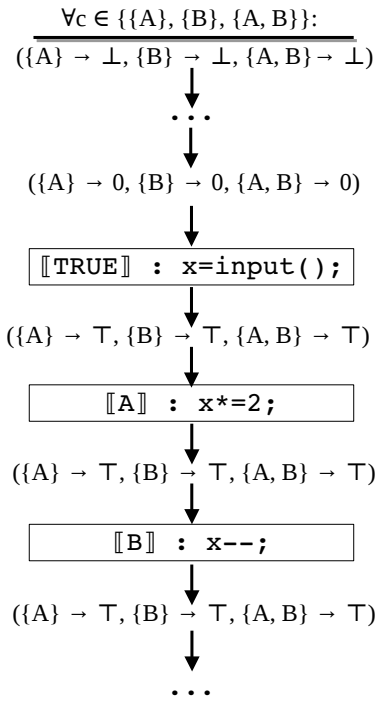


Figure 3.4: Sign analysis' (partial) result, performed for the example in Figure 3.1, using Simultaneous Feature-Sensitive

In order to achieve static energy consumption analysis in a SPL it is necessary not only to determine such behavior, but also to understand how it can affect energy consumption. A more thorough analysis, similar to worst case execution time (WCET) analysis, is needed. In such an analysis, the goal is to give an upper bound for each instruction, determine how they all are related, and calculate an estimate for the worst case scenario.

We present, in this section, our strategy for determining an energy consumption worst case scenario based on the classic WCET analysis [Wilhelm et al., 2008]. This strategy is composed of 4 phases:

A. use fixed-point computation to gather information about how instructions' execution influences the hardware energy consumption behavior;
B. use a data-flow analysis to determine loop upper bounds;
C. at each program point, an energy model calculates an energy-bound for every product of the SPL;
D. finally, use all the information from previous steps, a constraint solver computes a global energy-bound for all the products of the SPL.

3.3.1 Static Prediction of Energy Behavior

In classic execution time prediction, the first step towards determining the WCET is to analyze how the processor will behave when executing a certain statement [Wilhelm et al., 2008]. This allows us to determine how the processor will behave while also depending on what statement(s) was(were) executed before. This is called the Processor Behavior Analysis step.

The goal is to create an abstract model of such behavior, with all the processor states which can affect energy consumption. In other words, to create a finite lattice where the elements are those states, and the necessary conditions to switch state are described by monotone transfer functions.

Having such an abstract model for the processor's energy behavior, all the properties from static dataflow analysis (as presented before) hold. This was already proven to work before [Møller and Schwartzbach, 2015]. For our purpose, we want to make it possible to use this approach to model every hardware component, in every possible state, which influences the energy consumption.

To fit static program analysis requirements, each component's behavior must be independent of all others. Nevertheless, the behavior analysis must be done only once. We need to define a prediction model, $\mathbb{P}$, able to determine how the hardware components behave when executing a given instruction. The model $\mathbb{P}$ consists of a set of transfer functions. The input of the prediction model $\mathbb{P}$ is a list of n lattice elements, $\mathbb{S}$, one for each hardware component, and the output is $\mathbb{S}'$, corresponding to a list of new lattice elements. When analyzing a statement, the elements in $\mathbb{S}'$ represents the state of the components **after** the statement is executed.

As shown in Section 3.2.3, the analysis in a SPL must be extensible to every product. This requires two things: (1) for every node/program point there must be a list of lattice elements for every possible product, and (2) for a given product $\mathfrak{p}$ the corresponding list of lattice elements must be updated (using the transfer functions) if and only if the instruction in the node is included in product $\mathfrak{p}$.

The result of an analysis such as this one is similar to the one presented in Figure 3.4. However, each node will have associated a map between product configurations and a list of n lattice elements (for n hardware components), instead of only one lattice element.

3.3.2 Data-Flow Analysis

In the context of WCET, data-flow analysis is used for gathering information about the possible set of values for a variable at various program points [Wilhelm et al., 2008]. This is also a key component for our approach, since it enables us to determine the variables' values before/after the execution of each iteration of a loop. We can use such information to test the loop condition after every iteration, and determine the maximum number of iterations: the loop upper bound.

In WCET analysis all loops must have an upper bound. In fact, determining a loop bound is crucial to increase the accuracy of the analysis. However, it is often impossible to accurately determine such a bound. In order to address this problem, usually additional information is provided by the programmer: a range of values that input variables may have.

Assuming that such information is provided, we developed a technique capable of determining the upper bound of any loop. This technique is the result of combining static program analysis to determine the variables' values at each program point [Møller and Schwartzbach, 2015], and abstract execution to automatically derive loop bounds [Gustafsson et al., 2006].

Figure 3.5 expresses the result of our technique in Example 3.1 at every program point until the first iteration of the loop concludes. We can see that the analysis is feature sensitive, i.e., the propagation of a variable's abstract value in an assignment is only considered for products which include such assignment.

Our approach follows the forward analysis principle [Møller and Schwartzbach, 2015]. It begins at the program's first instruction, and once it reaches an assignment the abstract value for the variable is updated. The result is propagated to the following instructions, and so on. Considering, for example, that the instruction being analyzed is an assignment of the value 5 to a variable a. The abstract value for a will be 5, and the successor instructions will consider that value for variable a.

Our technique requires a map structure, M, which relates variables and a range of possible values for them. Every time an instruction is found that assigns an input value to a variable, the map structure M is consulted and the variable gets an abstract value representing its range of possible values. If there is no entry in the map, the variable gets the abstract value $\top$, which will be considered as the maximum possible value.

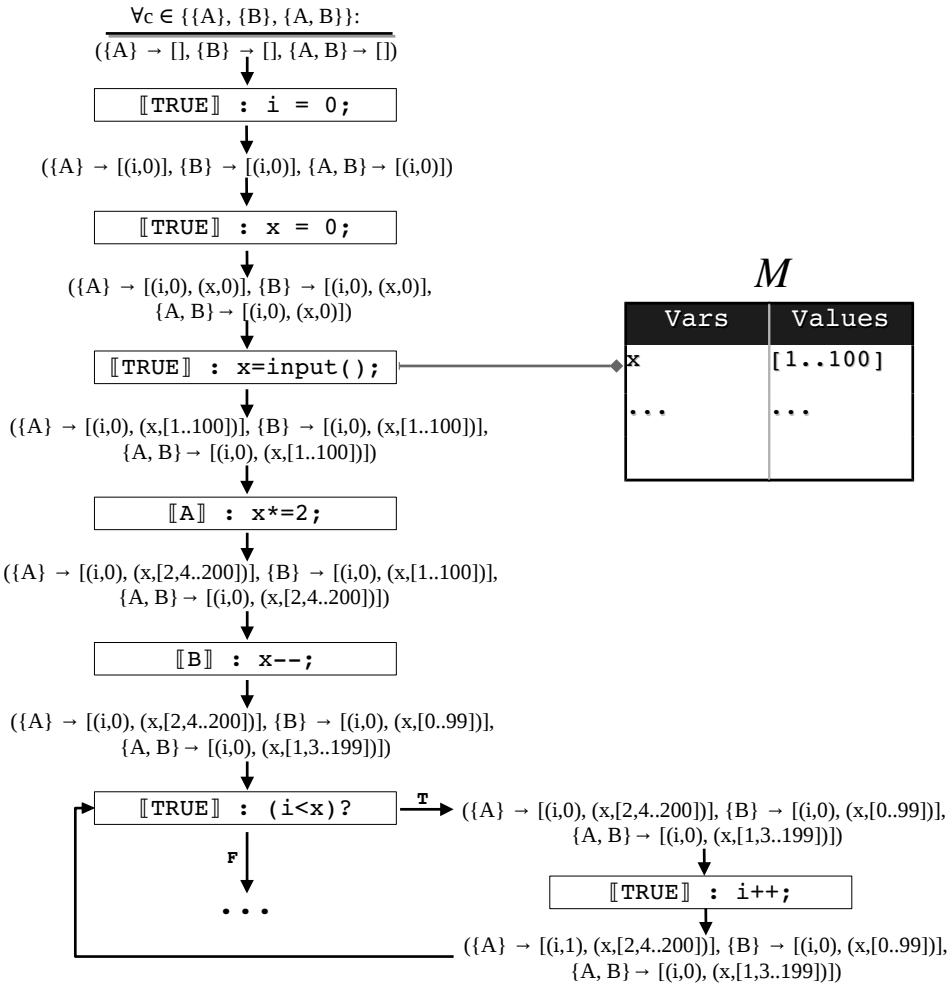


Figure 3.5: Example 3.1 data-flow analysis (loop's first iteration.)

If a given program point has more than one predecessor instruction (e.g., at the end of if structures), it means the variable abstract values must be **joined**. We defined a join operator that merges the possible variable values. For example, if the instruction `a++` has two predecessors that propagated the abstract values 1 and 5 for variable a, then the join operator transforms those values in a single one: $[1,5]$. This means that variable a can have two different values at that entry point, which is in fact true. The propagated result after the instruction `a++` will then be $[2,6]$.

Once the analysis reaches a loop, it knows all the possible abstract values for those variables, and tries to symbolically execute such loop with them. For every iteration of the loop the stop condition is verified, and if it fails for **all** possible values of the variables, then we have reached the upper bound of that loop.

To ensure that this analysis will finish, it is necessary to provide an exagerated upper bound, for the case that the loop condition never fails for the possible abstract values (when the map structure M does not have an entry to every variable with input values assigned to it). This value can be parameterized, according to the program under analysis. For example, for a program calculating the length of a word, this value should be the maximum length of any word under consideration.

3.3.3 Combining SPL Static Analysis With Energy Estimation

In Section 3.3.1, we explained how to predict the behavior of hardware components in a feature-oriented manner. This analysis allows us to predict how the hardware will behave after executing every statement, and for every valid product in the SPL.

Similarly to SPL static analysis, the result will be stored in a CFG with a lifted lattice $\mathscr{L}$ in each node, representing the hardware components states for each product, as shown in Figure 3.6. Here, every $\mathscr{L}_i$ represents a list of n lattice elements, one for each of the n hardware components considered. For example, $\{A\} \rightarrow \mathscr{L}_1$ (where $\mathscr{L}_1 = \{\mathtt{l}_1, ..., \mathtt{l}_n\}$) means that $\mathscr{L}_1$ holds the states of the n hardware components after the execution of `x=input()`, for the product with the configuration $\{A\}$. After executing `x*=2`, the lifted lattice $\mathscr{L}_1$ will change to $\mathscr{L}_2$ for products with configuration $\{A\}$ and $\{A, B\}$, since that instruction is only included in products with feature A.

The information in every node represents only the state of the machine **before** executing the instruction in that node. In order to get an energy estimation, and following the WCET principle, we need to match the states (lattice elements) with an energy model, $\mathbb{E}$, where the consumption per state is specified. $\mathbb{E}$ can be described as a function that takes as argument a tuple (i, C, s), where i is the node instruction, C is the hardware component, and s is the component state (i.e., the lattice element), and returns a consumption value reflecting the work made by the hardware component C to execute i while in the state s. With n hardware components, this function must be invoked n times for each node.

The energy model will give a local energy bound for each node in the CFG, which will serve as input to a constraint solving system capable of predicting the energy consumption of the entire program in the worst case scenario based on such local bounds. This will be further explained in

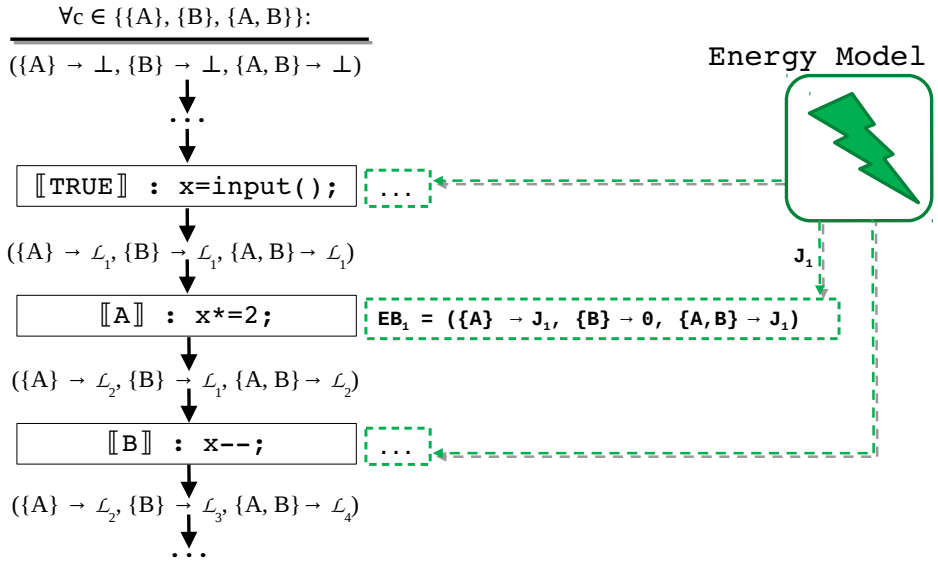


Figure 3.6: Calculating local energy bounds for each node in the CFG

Section 3.3.4. For the purpose of validating our work, we created a specific energy model, which we will present in Section 3.5.1. Our analysis can, however, use different energy models provided by, for example, hardware manufacturers.

After matching all this information with the aforementioned energy model the CFG will have, for every node i, a set of local energy bounds EB_i. Each element j of every EB_i will be the local energy bound for a valid product configuration j. If the instruction in the node is not included in the product configuration j, then its local bound will be 0.

Figure 3.6 represents part of the CFG of Example 3.1, where it is possible to see the energy model assigning an energy value to all product configurations in a node. In the example, since the instruction is included only in products $\{A\}$ and $\{A, B\}$, they both receive the value J_1, while the product $\{B\}$ gets 0. Also, the value J_1 reflects the combined consumption of all hardware components, and so it depends on the list of lattice elements represented by $\mathscr{L}_1$.

3.3.4 Worst Case Prediction

The primary goal of our work, as we said before, is to determine the energy consumed by every product in a worst case execution scenario. Until now, we have shown how to determine the energy consumption of each statement individually, concerning the context in which it executes. However, we need to estimate the overall consumption of a product, considering that not all state-

ments have the same impact, for example loop statements will probably execute more frequently, and if branches may not always execute.

There is a widely used approach in WCET prediction called IPET - Implicit Path Enumeration Technique [Li and Malik, 1995]. This technique consists of translating the CFG into a system of constraints. Such constraints are a result of combining program flow and CFG node execution time bounds, with the program WCET estimation being the result.

From static analysis, each node i has a local execution time upper bound, t_i, expressing the contribution of the node's statement to the total execution time, when executed once. IPET considers a new variable, x_e, where e is an entity which can be a node or an edge between two nodes. This variable represents the number of times e is expected to execute, and all these count variables are subject to constraints reflecting the program's structure and possible flows.

The constraints for IPET must follow a basic but strong principle: the number of times an entity e is accessed, x_e, **must** be equal to the sum of the number of times each of its predecessors are accessed, while also being equal to the sum of the number of times each of its successors are accessed. In other words, for every entity e,

$$x_e = \sum x_i = \sum x_j$$

for all entities i and j that are predecessors and successors of e.

For our purpose, the local upper bound for each node (t_i) will be the one determined by using the method described in Section 3.3.3, i.e. the energy consumption local bound. IPET will take care of determining the values for every x_i, in a way in which it maximizes the overall energy consumption. In other words, the worst case energy consumption $WCEC$ (just as WCET) is calculated by maximizing a function defined as the sum of all t_i multiplied by its corresponding x_i. In other words, with N instructions in a program, the WCEC function for a product is defined as:

$$WCEC = max(\sum_{i=0}^{N} x_i * t_i)$$

Figure 3.7 shows how the constraints for the product $\{A\}$ of Example 3.1 would look like. Note that the loopbound constraint indicates that the loop will execute at most 200 times, since the loop is upper bounded by the value of the variable x, and it's maximum value at that point is

200. For product $\{B\}$, this constraint would be $x_F \leq 99$.

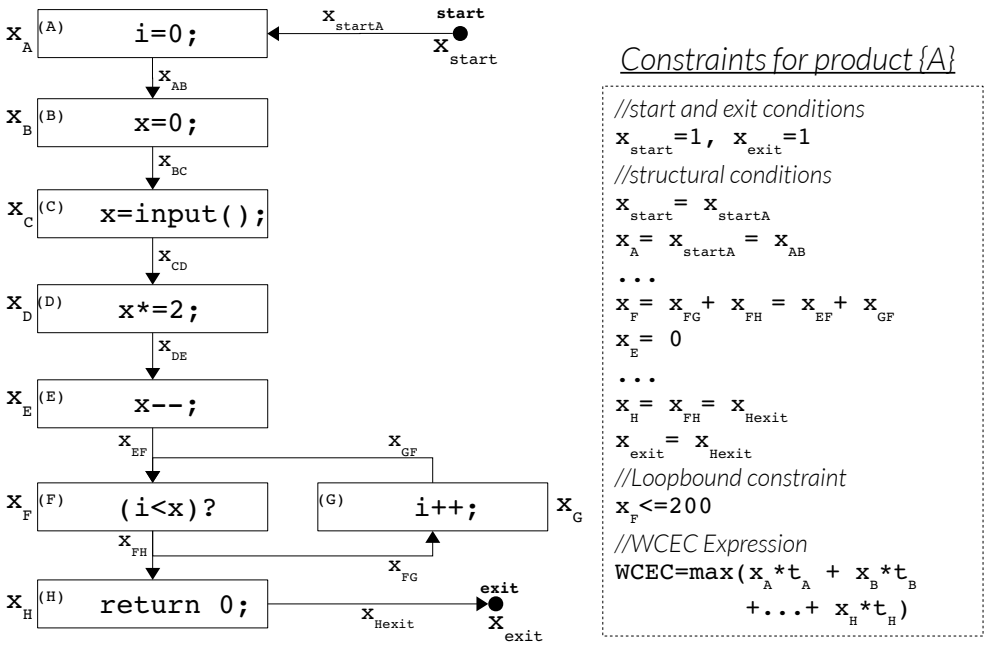


Figure 3.7: IPET constraints for product $\{A\}$ of Example 3.1

The IPET technique could already be used to determine the worst case energy consumption scenario of a product. However, we have to provide information about statements to be included/excluded in the product, since we do not want to consider the energy consumption influence of statements from excluded features. For this, we can set a few extra restrictions that will assure the exclusion of feature-dependent statements not included in the product being analyzed. This will be a restriction such as $x_e = 0$, for all entities e which are instructions and not part of the product being analyzed.

Using this strategy we may then calculate the worst case energy consumption scenario for each product in a SPL. Considering the 4 products included in Example 3.1 (with configurations $\{A\}$, $\{B\}$ and $\{A, B\}$), our technique determines the product with configuration $\{B\}$ is the most energy efficient. This is due to the fact that the loop for this product is bounded to 99 iterations, roughly 50% less than the loop bound for $\{A\}$ (200 iterations) and $\{A, B\}$ (199 iterations). This causes the amount of work to be substantially less in $\{B\}$, thus the energy consumption will also be lower.

3.4 Overview of the Prototype Tool

To validate our technique, we developed a prototype tool, named Serapis, which is thoroughly described in Section 6.3. This tool is able to, given the source code of a SPL, statically reason about the energy consumption of its products. Its overall workflow is depicted inf Figure 3.8.

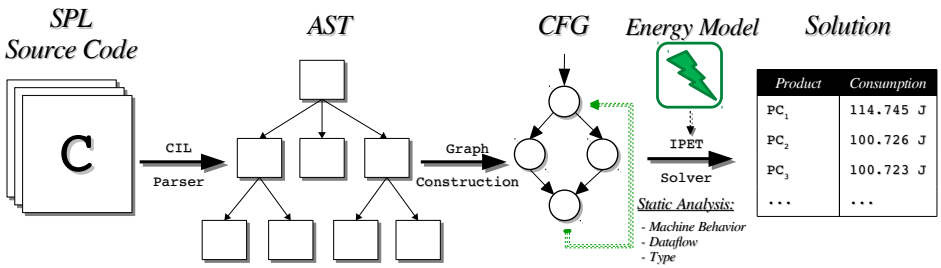


Figure 3.8: The Serapis Workflow

The workflow of Serapis is divided in 3 steps. The first step consists of creating the feature-sensitive CFG for the SPL, following the principles presented in Section 3.2. Our approach in this step was to create the CFG from the Abstract Syntax Tree (AST) representing the analyzed SPL. In order to simplify the source code analysis, we considered a simplified intermediate representation of the C language, called **C Intermediate Language - CIL** [2]. Hence, we parse CIL code into an AST representation, and then transform it into a CFG.

As explained in Section 3.2.3, every node in the CFG must contain information about a source code statement and the features that implement it. This means that, when parsing the CIL code, we need to use the preprocessor references as context management directives. Hence, when one such reference appears, we know if a new feature must be included/excluded from the next node.

The second step is the static analysis step, which corresponds to the first three phases of the strategy presented in Section 3.3. Here, the CFG is given as input to a series of functions, each one responsible for a specific type of analysis:

- **Energy behavior analysis**: determines the machine state of hardware components after each statement execution, using the fixed-point computation technique.

[2] More information can be found here: https://people.eecs.berkeley.edu/ necula/cil/ (visited on October 19, 2020).

- **Flow analysis**: determines the variable types and values before each loop execution, and symbolically executes each loop to determine its upper bound.
- **Local energy bounds**: considering the result of the two previous analysis, the energy model calculates a local energy bound value for each product, on each node.

The result of this analysis is passed to the third and final step: the overall energy prediction. In this step, the CFG and the loop upper bounds are used to create the constraints and the maximization function needed by IPET (as described in Section 3.3.4). Using an integer linear programming (ILP) solver, we can then obtain the output for this final step: the WCEC predictions for all existing products.

3.5 Experimental Evaluation

This section describes the experiments that we designed and conducted to evaluate the technique that we have proposed in Section 3.3 to statically predict the energy consumption of products in a SPL.

To support further experiments, we searched the literature for C-based SPLs with public source code. As a result, we managed to gather 4 SPLs suitable to be analyzed by our prototype tool, from 2 different sources: (i) 1 SPL from the disparity benchmark of the San Diego Vision Benchmark Suite [Venkata et al., 2009][3], and (ii) 3 from SPL2go[4], which is a repository of SPL case studies.

The disparity benchmark SPL consists of 7 products, all of which designed to calculate the disparity between two images: given a pair of images for a scene, taken from slightly different positions, the disparity map algorithm computes the depth information for objects jointly represented in the two pictures. This can be used to detect the depth of objects in an image. All of its 7 products share most of the source code, but differ on the algorithm to compute the disparity between 2 images.

The SPLs obtained from the second source are called Elevator, MinePump, and Email System, which have been used in several other SPL-related research works [Apel et al., 2011, Apel et al.,

[3]The products' source code is available at http://specs.fe.up.pt/publications/array16.zip (visited on October 19, 2020).

[4]SPL2go projects webpage: http://spl2go.cs.ovgu.de/projects (visited on October 19, 2020).

2013, Siegmund et al., 2013]. As the name suggests, Elevator simulates a control system for an elevator, with its variability expressed with features such as the inclusion of a weight detector or a priority floor; MinePump describes an interface for operating a mine pump system, where it is possible to have variations with/without, for instance, a methane alarm or a high/low water sensor; Email System is comprised of a solution for simulating sending/receiving emails, with the possibility of including, for instance, encryption/decryption features, or a forward functionality.

An overview of the 4 SPLs is depicted in Table 3.1[5]. For each SPL, we include the number of features, the number of products which are possible to generate, and the total number of lines of code in that SPL.

Table 3.1: Overview of SPLs used in the experiment

SPL	LOC	Features	Products
San Diego	2938	7	7
Elevator	877	6	20
MinePump	279	7	64*
Email System	258	9	40

We should emphasize that we are aware of the existence of other C-based SPLs also used in research works [Medeiros et al., 2016]. Nevertheless, the examples we selected were the ones that, to the best of our knowledge, fit the requirements to be analyzed by our prototype tool: (i) they are real-world software systems, widely used for research purposes, (ii) they avoid using features that somehow limit static analysis (e.g. recursive functions), and (iii) their code can be interpreted as CIL code (i.e., the intermediate C language that consider), so it can be parsed and transformed into a CFG.

For our analysis, we implemented a custom made tool for dynamically measuring the energy consumption of a product, in addition to the tool implementation that we described in Section 3.4. Then, we assess our technique by comparing the observed consumptions against our static predictions.

As we described in Section 3.3.3, our technique needs an energy model with energy consumption values predicted for single instructions. So, in Section 3.5.1 we start by describing our approach to generate an energy model which can be used by our technique.

[5] Minepump has features with no associated code, which for our tool meant considering only 32 of the 64 possible products.

All studies were conducted on a desktop with the following specifications: Linux Ubuntu 14.04 LTS operating system, kernel version 4.4.0-59-generic, with 6GB of RAM, a Ivy Brigde Intel(R) Core(TM) i5-3210 CPU@2.50GHz

3.5.1 Static Model Creation

The energy and machine behavior models are crucial components needed by our technique. The results' accuracy can in fact be enhanced with a robust hardware behavior model, but more than anything else they depend on the energy values' precision in the energy model.

These kind of models are always hardware and language dependent, which means that we need to follow a modeling approach previously presented and build our own models. In fact, modeling the hardware behavior requires exhaustive knowledge about it: how it was programmed, what are the possible execution states, what is necessary to transition to/from a state, etc. Since there is not, to the best of our knowledge, an adequate study showing the influence of hardware behavior on the energy consumption of independent instructions, for now we were only able to create a machine with a single state per hardware component.

In order to increase the prediction accuracy, properly modeling the machine state behavior is the adequate mechanism to do so. Still, we argue that it is possible to obtain accurate estimations without such model. In fact, a similar, yet not entirely static approach has been used before, which considered such model for a simplistic and controlled hardware [Grech et al., 2014, Liqat et al., 2014], and with highly accurate results.

Taking these aspects into consideration, we define the consumption of each instruction on the single machine state will be an average of several measurements. This will result in a lattice with only one element, and therefore the transfer function is the identity function, since the lattice element will not change.

We developed a dynamic framework, called C-LEM[6], capable of inferring estimations for each provided instruction. It was used to build and energy model that considers all instructions in the CIL language. A more in-depth explanation of how the tool operates and how it was developed is included in Section 6.2. C-LEM supports two types of instructions: (i) single-cost operations and (ii) API/function calls. At the time being, it provides energy estimations based only on the

[6] C-LEM code: https://gitlab.com/MarcoCouto/c-lem (visited on October 19, 2020)

consumption of the CPU hardware component, but can easily be extended to consider other components. For the purpose of our evaluation, we modeled the energy consumption for 18 functions from the C library, which is the set of functions used in all analyzed SPLs referred before.

3.5.2 Experimental Methodology

In order to evaluate the accuracy and performance of WCEC and Serapis, we have used them to analyze the energy consumption with the 4 previously described SPLs.

The first step of the evaluation process consisted of determining the actual worst-case energy consumption value for each product. In order to do this, we followed a brute-force approach: for a given SPL, we generated all of its products, instrumented their source code with energy measurement calls, and then executed each product 200 times. This was performed by our dynamic energy measuring framework, which is responsible for executing the products with the same input and measure the energy consumption. Since our goal is to determine the energy consumed in the worst case, we retrieved the highest value for every product in the 200 measurements obtained[7]. Just like for the energy model creation, we used RAPL to get the actual measurements, which minimizes the instrumentation overhead by simply measuring the energy before and after the program execution.

The aforementioned procedure was repeated for all 4 SPLs, which lasted for over 9 hours. Once it finished, we obtained the observed energy consumption in the worst-case scenario for each of the 99 analyzed products[8]. The next step was to compare these values with the estimation for each one. By combining the created energy model, as described in Section 3.5.1, with our technique (implemented in a tool, as described in Section 3.4), we were able to compute accurate estimates for the worst case energy consumption of every product. The complete set of results are included in Tables A.1, A.3, A.2, and A.4 (in the appendix), where we included, for each product in a given SPL, the observed WCEC value, the estimated one, and the difference between them (both in Joules and in percentage).

A summary of these values is depicted in Table 3.2. For each SPL, we calculated the 1^{st}

[7] The 200 measurements always had similar consumptions (with small standard deviation).

[8] The complete set of results is available online: https://marcocouto.gitlab.io/serapis (visited on October 19, 2020).

(**Q1**), 2^{nd} (**Q2**), and 3^{rd} (**Q3**) quartile values, along with the **maximum** value, for both the difference between estimated and observed WCEC and the estimation error percentage.

Table 3.2: Study Results

	Difference (J)				% Error			
	Q1	**Q2**	**Q3**	**Max**	**Q1**	**Q2**	**Q3**	**Max**
San Diego Bench.	3.620	6.060	10.09	12.79	5.61%	10.29%	12.74%	18.06%
Elevator	0.001	0.003	0.003	0.004	3.96%	18.28%	22.96%	27.45%
Mine Pump	0.018	0.023	0.030	0.042	13.08%	17.23%	22.31%	30.81%
Email System	1.181	1.686	1.866	2.549	81.46%	115.4%	129.8%	182.0%

WCEC Performance

Our static analysis technique works at the source code level. Thus, it can be seen as an interpreter which, at execution time, parses the code, builds data structures (ASTs and CFGs), and performs complex computations. As expected, inferring the WCEC analysis of a specific product will always take more time than executing its compiled (and optimized) code. Nevertheless, performing an adequate energy analysis dynamically implies that **(i)** each analyzed product needs to be generated, **(ii)** its code instrumented with energy monitoring code, and **(iii)** it must be exercised several times, in order to have a statistically conclusive analysis.

By following a static approach, our technique avoids this dynamic analysis "overhead" time. In fact, the importance of reducing this time is more notorious with a high number of products to be analyzed. For our case study, the SPLs dynamic analysis step, which consisted of executing each product 200 times and collecting energy measurements each time, took over 9 hours. Comparatively, it took ≈ 3 hours to perform a complete WCEC analysis for all 4 SPL: ≈ 14 min for the San Diego benchmark SPL, ≈ 80 min for Elevator, ≈ 1 min for MinePump, and ≈ 95 min for Email System. As such, it is easy to understand how one could benefit (in terms of time saving) from using our technique to analyze SPLs with a high variability level.

Nevertheless, we are aware that our technique can still go through a few performance improvements. For instance, regarding static analysis, the technique that we propose assures that the source code of every feature is analyzed only once, contrary to the use of traditional static analysis techniques. The exception is the worst case prediction technique shown in Section 3.3.4, which works at the product level. This means the solver will run once per product and thus re-

peat the calculations for shared features. In the future we intend to improve the performance of Serapis by encoding a SPL in the solver in a different way so it does not unnecessarily recalculate the constraints for shared features.

WCEC Accuracy

Having a reliable and accurate estimation was our main goal for WCEC. As we can see, the consumption values estimated by the technique were always higher then the measured ones, which is a crucial property to be maintained, since we want to predict the worst case. Upon examining the results for the first 3 SPLs, we see that the difference between estimated and observed values is, in percentage, considerably low. If we consider the median value (**Q2** column), this is always below 20%, and only 1 SPL had a maximum error percentage of 30%. Moreover, the **Q3** values for these SPL are always below 23%, which means that, for 75% of the analyzed products, the estimation was quite accurate. These are promising results, specially if we consider that this is the first approach for energy consumption prediction in SPLs, and that this proof-of-concept approach still has a few naive components, and therefore potential for improvements.

Nevertheless, the results for the last SPL (Email System) revealed that our technique inferred an excessive WCEC overestimation. Considering all of its 40 products, the estimated WCEC is expected to be 66.5% higher than the observed one in the best case scenario, and it can be up to 182% higher in the worst case scenario. These values revealed that, although the main property is still maintained (overestimation in favor of underestimation), there is a significant lack of precision by the technique in this specific case. Therefore, a high margin for improvement was exposed here, which we thoroughly addressed and will explain in the next section.

3.5.3 Accuracy Enhancements

After examining the obtained results for the Email System SPL, we sought to understand what was causing the excessive overestimated WCEC values. In order to systematically address the high error percentages observed previously, we have conjectured two possible causes for it: i) that our technique was inferring an execution path for the products that was entirely not the path that such products followed when being tested/executed and ii) that our model overestimated the energy consumption for some instructions.

```
"features": [
   "ENCRYPT",
   "DECRYPT",
   "KEYS"
 ],
"estimated"   : 3.139435,
"instructions": {
    "ARR_POS_LOAD"    : 9700025,
    "ASS_VAR_VAR_CHAR": 3600006,
    "ASS_VAR_ADDR"    : 1800010,
    "FUN_sprintf"     : 200000,
    "FUN_fputs"       : 600000,
    ...
}
```

Listing 3.2: WCEC execution path output

Tool Extension and Assessment

In order to assess the first conjecture, we extended Serapis. This extension was achieved in such a way that, for each product, not only the estimated WCEC value is provided as output, but also a full characterization of the WCEC execution path, containing the number of times each modeled instruction is executed in it. For EmailSystem, an example of such output for a product containing the features `ENCRYPT`, `DECRYPT`, and `KEYS` is partially depicted in Listing 3.2.

We can see that this product had an estimated WCEC of around 3.14 J, and among the instructions responsible for such consumption were ≈ 9.7M memory loads of array elements, 3.6M assignments from/to variables of type `char`, 1.8M pointer assignments, 200k calls to the `sprintf` functions, and 6000k calls to the `fputs` function.

Having extended Serapis, we conducted a systematic code inspection procedure, in which we have manually collected the expected modeled instructions for a product's WCEC execution path. These instructions were then compared against the (new) output from Serapis, an exercise that was performed for 5 randomly selected products within EmailSystem.

We believe that the results from our analysis provided enough evidence for us to refute our first conjecture, and to further explore the second.

Model Refinement

As our WCEC prediction provided accurate results for all the considered SPLs except `EmailSystem`, we followed the assumption that instructions with overestimated energy consumptions are only present in `EmailSystem`. This led us to delve deeper into its 40 products: for each product, we have inspected its execution path in search for instructions that were being considered in **all** 40

products, and which were not considered in **any** of the products of Elevator and MinePump.

We found 6 function calls (`puts`, `sprintf`, `strcmp`, `strcpy`, `fprintf`, and `strcat`), along with 3 other instructions (casting to a char, casting to a pointer, and assigning a char variable to another variable) that fit this criteria.

Our analysis also revealed that these 9 instructions represented between ≈ 30% and 40% of the WCEC execution path, depending on the product.

We then analyzed if these instructions were being considered in the WCEC execution path of any of the 7 products from the San Diego Benchmark SPL. We found that these instructions represented only ≈ 1% of the WCEC execution path, in any of the 7 products. From this observation, and following the issues in Section 3.5.1 that arise from modeling functions that depend on their arguments length/size, we can correctly assume that: i) there was a clear overestimation for each of the 9 instructions; and ii) updating the estimation for them will not compromise the predicting ability of the model.

In order to decide which values to update, and by how much, we used a well-known optimization technique called Simulated Annealing [Kirkpatrick et al., 1983]. This technique is an iterative method, used with the purpose of finding an accurate approximation of the global optimum of any function. This technique expects a list of variables to update, a function to randomly update the variable's values, and a cost function which, for each iteration, determines if the ideal solution is closer or not after updating the variables. Our update and cost functions are shown in Listing 3.3.

In every iteration of our Simulated Annealing algorithm, the update function receives the 9 energy model variables (corresponding to the 9 overestimated instructions), and randomly decides how many of them it will update. Then, for each time, it will randomly decide which variable will suffer an increment/decrement, and how big should this `delta` value be. As line 15 in Listing 3.3 demonstrates, the `delta` value comes from the result of a modulo operation between random value inside the interval $]0, 1[$, and variable's previous value divided by a number between 2 and 10. The variable's last value can be converted to a `default` value (10^{-12}, which in this case is $1/100$ of the lowest value in the energy model), in order to assure that a given variable's value never reaches 0 or less. With this approach, we ensure that: (1) the decision on whether update a variable or not is fully random, and (2) a variable will always suffer a small, yet completely random, increment or decrement. These criteria are necessary requirements of the Simulated

```
def update(vars):
    frac, default = rand(2, 10), 1E-12
    new_vars = vars
    # how many changes per iteration? random.
    # between one and half the length of vars.
    num_changes = rand(1, len(vars)/2)
    for i in range(num_changes):
        # which var to update? random.
        var = new_vars[rand(0, len(vars) -1)]
        old_val  = max(var.value(), default)
        # how much should it increase/decrease? random.
        sig   = rand([-1, 1])
        delta = (rand() % (old_val/frac)) * sig
        var.set_value(old_val + delta)
    return new_vars

def cost(model, products):
    cost = 0, threshold = 0.05
    for p in products:
        p.estimate_energy(model) # update estimation
        est, obs = p.get_estimation(), p.get_observed()
        error    = (est - obs) / obs
        if error >= 0:
            cost += max(0, error - threshold) ** 2
        else:
            cost += (error * 10) ** 2
    return cost
```

Listing 3.3: The update and cost functions used by Simulated Annealing

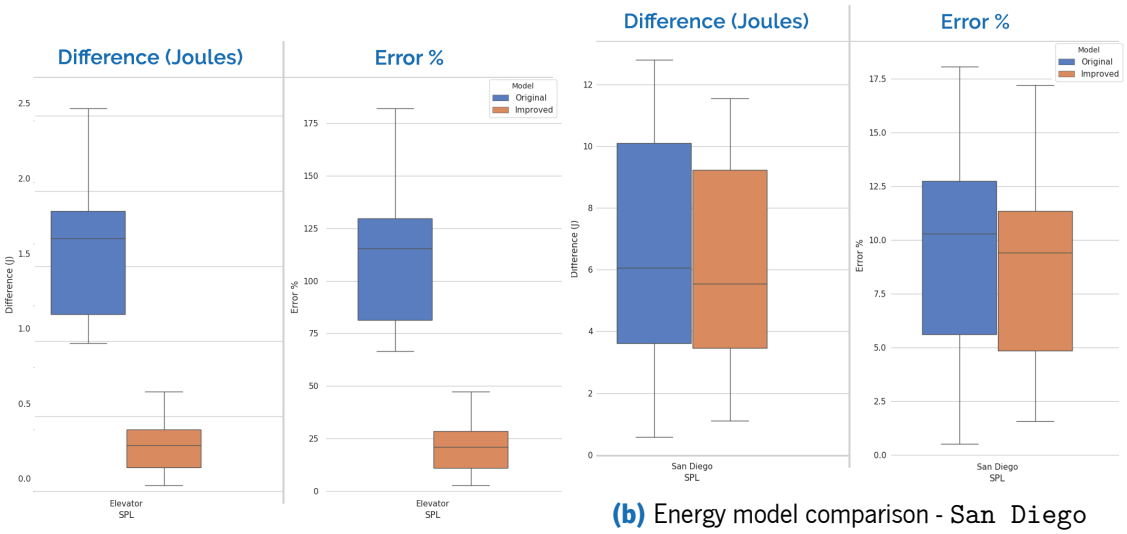
Annealing technique.

Simulated Annealing decides, on every iteration, whether or not the computed values should be used to update the energy model, or reject them and keep the old model. This decision will be mainly influenced by the result of the cost function. If the new model has a lower cost, it will replace the old one. If not, a probabilistic method will be used to determine if the model should be replaced or not. To fit our needs, we defined the cost function as seen in Listing 3.3: (1) for each product, we use the new model to compute a new WCEC estimation; (2) we calculate the error percentage between the estimated and observed WCEC; (3) if it is positive, it means that the estimation exceed the observation, and we add to the overall cost the square of the error percentage minus a 5% threshold (to benefit models with similar lower errors among products), or zero if the error is below 5%; (4) if the error is below 0, the cost should be much higher because we never want the estimation to be lower than the observation.

Improved Results

The Simulated Annealing algorithm took about 5 minutes to execute, and we have reached a new energy model, which we have applied to all 4 SPLs.

As expected, the estimations for all products within Elevator and MinePump suffered no alterations whatsoever. For EmailSystem, however, the results were significantly better, and for San Diego Benchmark the results were marginally better, as shown in Figure 3.9.



(a) Energy model comparison - `EmailSystem` SPL

(b) Energy model comparison - `San Diego Benchmark` SPL

Figure 3.9: Accuracy comparison: Original model vs Simulated annealing-improved model

Every quartile threshold, as well as the max/min observed values for difference (in Joules) and error percentage, is lower for the improved model. In fact, the highest accuracy error observed with the improved model is lower that the lowest accuracy error obtained with the old model. Moreover, the `San Diego Benchmark` SPL, whose results had already been shown to be accurate, also revealed overall improvements, since all quartile thresholds, as well as the maximum value, decreased with the improved model.

Nevertheless, the minimum value increased a little, which tells us that there was one product whose predicted WCEC was actually less precise using the improved model. We found it to be the first product, which had an error percentage of $\approx 0.5\%$, and with the improved model it is $\approx 1.55\%$. It reflects a difference increase of ≈ 1 J, in a product with an observed WCEC of 114 J, and was the only product among the examined 99 which WCEC estimation accuracy has decreased. Hence, we consider this to be a negligible precision loss.

3.6 Threats to Validity

Our work is focused on developing a technique for static energy consumption prediction of SPLs, and validate it with as a solid proof-of-concept. Nevertheless, we identified a few aspects that may affect the validity of our work, and the applicability of our technique.

As explained in previous sections, the current version of our tool requires that the complexity of the SPLs analyzed is somehow limited: recursive functions are not supported, and the code must be compliant with the CIL syntax. We should emphasize that these limitations are not technique dependent, but rather tool dependent. Since this work is focused on developing a proof-of-concept, rather than a fully generic tool, not addressing recursions and choosing CIL as supported language helped us minimize the implementation effort, while still enabling us to test our technique on real-world SPLs. However, this ultimately meant that we were also limited in the number of SPLs that could be used for the validation process.

In order to address the CIL limitation, it is necessary to provide support for a C-based AST representation of SPLs. Such representation would inevitably be much more complex, it would require an additional effort to transform it into a CFG, and the energy model should also be extended. Recursive functions, on the other hand, would need to be expressed as loops in the CFG; this requires an extra step to detect such functions, and properly include them as loops in the CFG. With these enhancements, the general applicability of our technique would in fact be increased, although we argue that the results for the potentially new SPLs would still be at least as accurate as the ones we obtained.

As we described before, our target was a proof-of-concept technique that could deliver accurate estimations. As such, we endowed the technique with mechanisms for potential additional accuracy enhancements, such as the machine behavior model. Although their impact on the estimations accuracy in not fully addressed in this work, we believe that, with the highly complex hardware components used nowadays, studying and building such mechanisms would require a work of its own. Moreover, a few works have proved that using similar mechanisms for static energy analysis, although applied in different contexts, can only help to improve the accuracy of the estimations [Grech et al., 2014, Liqat et al., 2014]. Hence, we argue including them in our technique, regardless of their simplicity, is deemed relevant.

3.7 Conclusions

This chapter described a methodology for statically predicting the energy consumed by all products within a SPL, in their worst-case execution scenario. The initial approach mainly relied on static program analysis applied to worst-case prediction, combined with an energy consumption model. We explained the full methodology for (i) creating the energy consumption model, as well as (ii) how to perform WCEC-tailored static analysis for SPLs, in both ways following existing state-of-the-art approaches.

The preliminary evaluation produced promising results for a real-world SPL, showing that it is possible to statically estimate how high the energy consumption of a product would be, given a particular usage scenario. Building on these findings, we further assessed our approach using 3 new SPLs widely used in research. For 2 of them, the results still showed a high prediction accuracy. Nevertheless, the WCEC values for the remaining SPL were quite overestimated.

After an update to Serapis, and a thorough analysis procedure, we were able to understand that the energy model was overestimated for some instructions, which were extensively used in the products with overestimated WCEC. We adapted an optimization algorithm called Simulated Annealing in order to automatically decide how to update the overestimated values. The obtained model revealed to be effective, resulting in improvements for 2 of the 4 analyzed SPLs. Considering the 99 products distributed through the 4 SPLs, we obtained a mean error percentage of 17.31% and a standard deviation of 11.19%.

The initial results obtained for the EmailSystem SPL revealed that the energy model, which is a crucial component of our technique, could be subject to additional improvements. We observed that, for an instruction with a variable type or number of arguments, overestimating the consumption to the highest observed value can in fact result in significant accuracy losses. Hence, it might be worthy to redefine how these instructions are expressed in the energy model. Considering the static analysis and IPET limitations, the most adequate way to do that would be to provide, for each of such instructions, multiple entries in the model, one for each variation that could be detectable in the static analysis steps.

In our studies, we have considered all the realistic CIL supported SPLs with publicly available source code that we could find in the literature. While we argue that this is a representative set,

we admit that other SPLs may potentially provide evidence of other improvement opportunities.

The accuracy of our technique relies mostly on the energy and machine behavior models, rather than the remaining static analysis components. Therefore, the models we provide are generic to any target device on which an C-based SPL could execute, yet quite simplistic. The machine behavior model does not reflect the real behavior of state-of-the-art CPUs, and considering a different behavior model would imply that the energy model should be not only instruction oriented, but also machine state oriented.

Nevertheless, we do believe that more complex and detailed models could be built and in other widely used contexts, particularly in the embedded systems realm. Here, C-based projects and SPLs are very common, and as such we aim at applying our technique is this scope.

4

Energy Code Smells in Android

Improving the energy efficiency of mobile applications is a timely goal, as it can contribute to increase a device's usage time, which most often is powered by batteries. Recent studies have provided empirical evidence that refactoring energy-greedy code patterns can in fact reduce the energy consumed by an application. These studies, however, tested the impact of refactoring patterns individually, often locally (e.g., by measuring method-level gains) and using a small set of applications.

We studied the application-level impact of refactorings, comparing individual refactorings, among themselves and against the combinations on which they appear. We use scenarios that simulate realistic application usage on a large-scale repository of Android applications. To fully automate the detection and refactoring procedure, as well as the execution of test cases, we developed a publicly available tool called Chimera.

Our findings include statistical evidence that i) individual refactorings produce consistent gains, but with different impacts, ii) combining as much refactorings as possible most often, but not always, increases energy savings when compared to individual refactorings, and iii) a few combinations are harmful to energy savings, as they can actually produce more losses than gains.

We prepared a set of guidelines for developers to follow, aiding them on deciding how to refactor and consistently reduce energy.

4.1 Introduction

Mobile devices such as smartphones or tablets are pervasive in our personal and professional activities, which is actually drawing significant attention for them to be energy efficient. Indeed, consumer satisfaction regarding a smartphone is highly dependent on its battery uptime [Fri, 2019]. For developers, battery uptime is also crucial, as abnormal drainage frequently justifies bad reviews in app stores [Khalid et al., 2015]. Finally, the amounts of energy spent by mobile devices are heavily affecting sustainability [Jha, 2018].

In the context of Android - the largest mobile ecosystem - several works have focused on documenting energy-greedy programming patterns and on finding better alternatives for them. In fact, identifying and refactoring such code patterns to improve energy consumption has already presented promising research results [Palomba et al., 2019, Morales et al., 2018, Cruz and Abreu, 2018, Carette et al., 2017, Cruz and Abreu, 2017, Jiang et al., 2017, Li and Halfond, 2014, Vekris et al., 2012, Linares-Vásquez et al., 2014, Li et al., 2016]. These results, however, have essentially been validated by testing code patterns individually and often in a small set of applications (sometimes only in one).

In this chapter, we consider 11 energy-greedy code patterns obtained from the literature, described in detail in Section 4.2. We conduct a study over a large-scale repository of 600+ Android applications to understand the frequency of occurrence of such patterns. Within the 200+ applications where the patterns were detected, we studied the impact that replacing them, individually and combined, by their documented alternatives has on the energy consumption. Moreover, as we consider all the possible combinations of the individual patterns, this resulted in 400+ refactored applications under analysis.

To perform our study, we developed an extensible, fully automated framework called Chimera, which is able to detect and refactor the patterns. Each pattern is considered individually and is also combined with all the other patterns. Chimera also measures the energy consumed by an application in different simulated usage scenarios, before and after refactoring.

In summary, the main contributions of this work are:

1) An analysis of how energy-greedy patterns proposed in the literature are distributed over a large-scale repository of Android applications. This is described in Sections 4.3 and 4.4;

2) A reusable prototype of a pattern-oriented testing framework (Chimera), described in Section 5.3, for the detection, filtering, and refactoring of patterns in Android applications; it can also run a set of usage scenarios on such applications, while collecting metrics such as energy consumption;

3) An empirical study, described in Section 4.5, to assess the energy impact of applying refactorings. We analyze, for each code pattern and combination of patterns, the test results for the Android applications on which they occur, and compare the obtained gains between each pattern/combination.

Using the results of the empirical study referred in **3)**, we aim at answering the following research questions:

Research Question 4.1

Do refactorings consistently lead to energy savings?

We aim at discovering if refactoring any individual pattern only results in energy savings, when considering a large set of different applications and execution scenarios. If that is not the case, than do the savings consistently surpass the losses?

Research Question 4.2

Do all individual refactorings lead to energy savings of the same magnitude?

With this, we want to know if there are two or more patterns that, when individually refactored, lead to similar energy savings. In addition, we also want to statistically determine what are the refactorings that lead to the higher energy savings overall.

Research Question 4.3

What are the refactorings that, individually or when combined, produce the higher energy savings?

Assuming that the savings associated with refactorings are statistically different among them, we want to fairly compare them in order to discover what are the refactorings (or combinations) where these savings are consistently higher.

Research Question 4.4

When refactoring for energy efficiency, what approach should developers follow?

The answer to this question will be targeted for developers and practitioners. In other words,

we want to provide a clear guideline to help decide when to refactor one (or more) patterns, using the findings that resulted from answering the previous research questions.

In line with the literature, our findings confirm (in the large and in the wild) that refactorings consistently lead to energy improvements. Complementary, we have found statistical evidence that combining refactorings can produce higher energy savings, although in a few cases we discovered the opposite. As a final contribution of our work, we present a set of guidelines that developers can follow when aiming at refactoring their applications to consistently improve their energy efficiency. All the study subjects, artifacts and results used and/or produced in our work are available in an online appendix[1].

The remaining sections of this chapter are organized as follows: Section 4.4.2 discusses the technique used for automatically refactoring energy-greedy patterns; Section 4.6 discusses threats to the validity of our study; Section 4.7 presents our conclusions and future work.

4.2 EGAPs - Energy Greedy Android Patterns

We have searched the literature for code patterns which have been tested for Android and that have proven to be energy inefficient; by an energy-inefficient pattern we mean a pattern for which an alternative exists, one that preserves the application's functionality, while consuming less energy. We have found 9 independent works that identify 11 energy-inefficient patterns and that propose alternatives for them [Palomba et al., 2019, Morales et al., 2018, Saborido et al., 2018, Cruz and Abreu, 2018, Carette et al., 2017, Cruz and Abreu, 2017, Jiang et al., 2017, Li and Halfond, 2014, Vekris et al., 2012].

In the context of our work, and in its description in this chapter, we shall refer to such a pattern as "**E**nergy **G**reedy **A**ndroid **P**attern", or **EGAP**.

In this section, we include a brief description of every EGAP we found, where for each one we indicate which research work(s) detected the pattern, while explaining what makes it energy greedy, what is the suggested alternative for it, and why such alternative consumes less energy.

[1] Online appendix URL: http://marcocouto.gitlab.io/android-egaps (visited on October 19, 2020)

4.2.1 EGAP #1 - `Draw Allocation`

This is the first of five EGAPs whose energy impact analysis was included in [Cruz and Abreu, 2017, Cruz and Abreu, 2018]. The authors aimed at understanding how fixing code patterns detected by Android lint[2] can improve energy efficiency. Lint's issues are divided into categories, such as Performance or Security. `Draw Allocation`, as well as the EGAPs described in Sections 4.2.2, 4.2.3, 4.2.4, and 4.2.5, is a **Performance** issue.

`Draw Allocation` occurs when new objects are allocated along with draw operations, which are very sensitive to performance. In other words, it is a bad practice to create objects inside the `onDraw` method of a class which extends a `View` Android component, as we see in the following snippet:

```
1  public class CloudMoonView extends View {
2
3    @Override
4    protected void onDraw(Canvas canvas) {
5      RectF rectF1 = new RectF(); ✘
6      ...
7      if(!clockwise) {
8        rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
9        ...
10     }
11   }
12 }
```

The recommended alternative for this EGAP, as of [Cruz and Abreu, 2017, Cruz and Abreu, 2018], is to move the allocation of independent objects outside the method, turning it into a static variable, as shown next:

```
1  public class CloudMoonView extends View {
2
3    RectF rectF1 = new RectF(); ✔
4
5    @Override
6    protected void onDraw(Canvas canvas) {
7      ...
8      if(!clockwise) {
9        rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
10       ...
11     }
12   }
13
14 }
```

[2] Lint is a code analysis tool, provided by the Android SDK, which reports upon finding issues related with the code structural quality. Website: developer.android.com/studio/write/lint (visited on October 19, 2020)

4.2.2 EGAP #2 - `Wakelock`

`Wakelock` is the second Android lint performance issue [Palomba et al., 2019, Vekris et al., 2012, Cruz and Abreu, 2018, Cruz and Abreu, 2017]. Basically, lint detects whenever a wake lock, a mechanism to control the power state of the device and prevent the screen from turning off, is not properly released, or is used when it is not necessary.

The following snippet shows an example of a wake lock being acquired, but not released when the activity pauses.

```
1  public class DMFSetTempo extends Fragment {
2
3    PowerManager.WakeLock wakeLock;
4
5    public void onClickBtStart(View view) {
6      wakelock.acquire();  ✔
7    }
8
9    @Override()
10   public void onPause() {
11     super.onPause();     ✘
12   }
13
14 }
```

The alternative here would be to simply add a `release` instruction as shown next:

```
1  public class DMFSetTempo extends Fragment {
2    PowerManager.WakeLock wakeLock;
3
4    public void onClickBtStart(View view) {
5      wakelock.acquire();    ✔
6    }
7
8    @Override()
9    public void onPause() {
10     super.onPause();
11     if (wakeLock.isHeld()) wakeLock.release();  ✔
12   }
13 }
```

4.2.3 EGAP #3 - `Recycle`

`Recycle` is another Android lint performance issue [Cruz and Abreu, 2017, Cruz and Abreu, 2018]. It detects when some collections or database related objects, such as `TypedArrays` or `Cursors`, are not recycled nor closed after being used. When this happens, other objects of the

same type cannot efficiently use the same resources.

The following snippet shows a `Cursor` instance being used without being recycled:

```
1 public Summoner getSummoner(int id) {
2   SQLiteDatabase db = this.getReadableDatabase();
3
4   Cursor c = db.query(TABLE_FAV, new String[] { ... };
5   ...
6   return summoner;  ✘
7 }
```

The alternative in this case would be to include a `close` method call before the method's return:

```
1 public Summoner getSummoner(int id) {
2   SQLiteDatabase db = this.getReadableDatabase();
3
4   Cursor c = db.query(TABLE_FAV, new String[] { ... };
5   ...
6   c.close();   ✔
7   return summoner;
8 }
```

4.2.4 EGAP #4 - `Obsolete Layout Parameter`

The fourth Android lint performance issue [Cruz and Abreu, 2017, Cruz and Abreu, 2018], `Obsolete Layout Parameter`, is the only one that is not Java-related. The view layouts in Android are specified using `XML`, and they tend to suffer several updates. As a consequence, some parameters that have no effect in the view may still remain in the code, which causes excessive processing at runtime. The alternative is to parse the `XML` syntax tree and remove these useless parameters.

The next snippet shows an example of a view component with parameters that can be removed:

```
1 <TextView android:id="@+id/centertext"
2   android:layout_width="wrap_content"
3   android:layout_height="wrap_content"
4   android:text="remote files"
5   layout_centerVertical="true"    ✘
6   layout_alignParentRight="true"> ✘
7 </TextView>
```

4.2.5 EGAP #5 - `View Holder`

`View Holder` is the last Android lint performance issue [Cruz and Abreu, 2017, Cruz and Abreu, 2018], whose alternative intends to make a smoother scroll in List Views. The process of drawing all items in a List View is costly, since they need to be drawn separately. However, it is possible to make this more efficient by reusing data from already drawn items, which reduces the number of calls to `findViewById()`, known to be energy greedy [Linares-Vásquez et al., 2014].

In order to better describe this EGAP, we introduce the following snippet:

```
1  public View getView(int p, View cView, ViewGroup par) {
2    LayoutInflater inflater = (LayoutInflater) context
3      .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
4
5    cView = inflater.inflate(R.layout.apps, par, false);
6    TextView txt =
7          (TextView) cView.findViewById(R.id.label); ❶
8    ImageView img =
9          (ImageView) cView.findViewById(R.id.logo); ❷
10   return row;
11 }
```

Every time `getView()` is called, the system searches on all the view components for both the `TextView` with the id "label" (❶) and the `ImageView` with the id "logo" (❷), using the energy greedy method `findViewById()`. The alternative version is to cache the desired view components, with the following approach:

```
1  static class ViewHolderItem {
2    TextView txtView; ImageView imgView;
3  }
4
5  public View getView(int p, View cView, ViewGroup par) {
6    ViewHolderItem hld; LayoutInflater inflater = ...
7
8    if (cView == null) {                              ❸
9      cView = inflater.inflate(...);
10     hld = new ViewHolderItem();
11     hld.txtView =
12           (TextView) cView.findViewById(...);       ❹
13     hld.imgView =
14           (ImageView) cView.findViewById(...);      ❺
15     cView.setTag(hld);
16   } else {
17     hld = (ViewHolderItem) cView.getTag();          ❻
18   }
19   TextView txt = hld.txtView;
20   ImageView img = hld.imgView;
21   ...
22 }
```

Condition ❸ evaluates to true only once, which means instructions ❹ and ❺ execute once, i.e., `findViewById()` executes twice, and its results are stored in the `ViewHolderItem` instance. The following calls to `getView()` will use cached values for the view components `txt` and `img` (❻).

4.2.6 EGAP #6 - `HashMap Usage`

This EGAP is related to the usage of the `HashMap` collection [Palomba et al., 2019, Morales et al., 2018, Saborido et al., 2018, Carette et al., 2017]. In fact, as stated in the Android documentation page, the usage of `HashMap` is discouraged, since the alternative `ArrayMap` is allegedly more energy-efficient, without decreasing the performance of `map` operations[3].

The alternative is to simply replace the type `HashMap`, whenever it is used, with `ArrayMap`.

4.2.7 EGAP #7 - `Excessive Method Calls`

Unnecessarily calling a method can penalize performance, since a call usually involves pushing arguments to the call stack, storing the return value in the appropriate processor's register, and cleaning the stack afterwards. This penalty was explored by [Carette et al., 2017, Li and Halfond, 2014], showing that the energy consumption in Android applications can be decreased by removing method calls inside loops that can be extracted from them. An example of an extractable method call would be one which receives no arguments, and is accessed by an object that is not altered in any way inside the loop.

The alternative is to replace the method call by a variable that is declared outside the loop, and is initialized with the return value of the method call extracted.

4.2.8 EGAP #8 - `Member Ignoring Method`

This EGAP addresses the issue of having a non-static method inside a class, and which could be static instead [Carette et al., 2017, Palomba et al., 2019], i.e., it does not access any class fields, it does not directly invoke non-static methods, and it is not an overriding method. Static methods are stored in a memory block separated from where objects are stored, and no matter

[3] ArrayMap documentation: http://bit.ly/32hK0y9 (visited on October 19, 2020).

how many class instances are created throughout the program's execution, only an instance of such method will be created and used. This mechanism helps in reducing energy consumption.

4.2.9 EGAPs #9, #10 & #11 - `Resource Leak`

`Resource Leak` [Vekris et al., 2012, Jiang et al., 2017] simultaneously refers to three EGAPs which are all particular cases of `Wakelock`, where a system resource is not properly released. Here, we consider the Sensor, Camera, and Media resources, which differ from `Wakelock` in the way resources are released. Since improving EGAPs on different resources can produce different energy gains, we decided to separate EGAPs #2 from #9, #10 & #11.

4.3 The Android Application Repository

In order to study the practical impact of replacing energy greedy patterns with their documented alternatives, a large-scale repository of Android applications from the real world was deemed necessary.

During our research, we came across large repositories of open-source Android applications such as the F-Droid catalogue[4]. However, using them in the context of our work would require additional effort in collecting the applications' source code: we would have to collect, for each application, the available information, parse it to find the source repository (which can be stored in different platforms, such as GitHub or GitLab), and download it. This additional step would be time consuming, and we wanted to avoid it.

Therefore, we followed an alternative approach to obtain a representative set of Android applications: we have taken the MUSE repository [Lopes et al., 2010, Lopes et al., 2017] as a starting point for our work.

MUSE[5] is a repository of Java projects, collected from public repositories contained in well known source code platforms, such as GitHub, Bitbucket or Apache. In addition to the projects' source code collection, MUSE has an associated database where, for each project, information regarding its source code is also available. For example, it is possible to know how many files

[4] F-Droid webpage: f-droid.org (visited on October 19, 2020).
[5] MUSE webpage: https://muse-portal.net/ (visited on October 19, 2020).

each project has, what import statements are used and what classes are declared in each file, what methods and variables are declared in each class, and so on.

Since Android applications are mainly Java projects, we were able to query MUSE and search for projects containing Android API components. The filtering strategy was to look at the import statements of each project. If there were any Android API imports, then it was and Android application project. We were able to obtain **609** buildable and executable Android projects, that we take as study subjects in this work.

In order to better understand the dimension of the Android projects repository, we computed, for each one of the 609 Android projects, the number of lines of code, Java files, XML files, classes, and methods. The results of this analysis are summarized in Table 4.1 where we include, for each metric, the maximum, minimum and average values.

Table 4.1: Applications Repository Metrics

	Java Files	XML Files	Classes	Methods	LOC
Min	2	1	2	1	22
Max	3,492	4,929	4,267	39,511	668,085
Average	73	407	65	524	10,822
TOTAL	**44,959**	**248,385**	**39,926**	**319,636**	**6,590,636**

4.4 Automatic EGAP Detection And Transformation On Android Code

In this section, we describe the methodology used for detecting and transforming the patterns described in Section 4.2, in the large-scale repository of Android applications described in Section 4.3. We start by explaining the techniques used to detect EGAPs in all the applications, and how we process the output from this analysis (Section 4.4.1). Next, we present our approach for the automatic transformation of such patterns, along with a discussion regarding its applicability (Section 4.4.2). Finally, we study how individual and combined EGAPs are distributed in the analyzed applications, and we present the final list of (individual and combined) EGAPs that we managed to analyze (Section 4.4.3).

4.4.1 Detection Methodology

Detecting a pattern in the source code of an Android application can be achieved by: i) parsing it into an abstract syntax tree (AST), ii) traversing through such AST, and iii) reporting where the pattern is found. These steps have already been abstracted within the lint tool which allows the possibility of creating rules for the detection of such patterns[6].

In fact, lint already includes rules supporting the detection of EGAPs **#1 - #5**. So, we needed to create custom rules for the detection of the remaining 6 EGAPs. Furthermore, the lint tool enables the selection of which rules to apply, which allowed us to consider the detection of "our" 11 EGAPs only, avoiding further analysis. Finally, it also contains a built-in report system which automatically produces an XML file indicating where a pattern is found in the code.

Given the advantages of using a lint-based approach for detecting EGAPs, we developed a new detection tool, called LintED[7], which is described more in-depth in Section 6.4. This tool consists of a collection of detection rules, which can report the existence of any of the 11 considered EGAPs.

Having at hand the custom rules for detecting all EGAPs, we then ran LintED on each of the 609 considered applications and collected the report files. The lint report consists of a list of EGAP occurrences within an application, with information regarding where in a file was the EGAP found. For each application, we parsed the report file to find which EGAPs were detected (and the number of occurrences). We group and store this information in `JSON` format.

In order to validate the accuracy of our detection methodology, we used a set of sample applications where all the patterns were manually injected, in all contexts on which they can occur[8]. After lint analyzed them, we manually searched for both false positives and negatives. We confirmed that all patterns were properly detected, with no inconsistencies found.

[6] More information can be found here: http://bit.ly/2Mh7aPF (visited on October 19, 2020).

[7] LintED webpage: https://gitlab.com/MarcoCouto/linted.

[8] For instance, for EGAP #6 we injected references to `HashMap` in variable types, method arguments, cast expressions, and instance creation statements.

4.4.2 Automatic Refactoring Approach

To automate the transformation of Android EGAPs, we reused the **AutoRefactor**[9] system: a well-known Java refactoring framework. This is an open-source Eclipse IDE plugin and it supports refactoring of Android applications, which was already used to refactor a small-set of energy-greedy Android patterns [Cruz and Abreu, 2018, Cruz et al., 2017]. In AutoRefactor a refactor is concisely defined as a set of source code transformation rules. Cruz et al. [Cruz and Abreu, 2017] defined these rules to express refactorings for EGAPs #1 - #5. We extend this work with transformation rules for the remaining EGAPs, and we integrate all refactorings in the command line interface (CLI) mechanism of Autorefactor.

Following the same approach as Cruz et al. [Cruz and Abreu, 2017], we designed the transformations to be generic without compromising the applications integrity. This means that although we detect all EGAPs reported in the literature that may occur in an Android application, it was not always possible to safely apply the corresponding refactoring automatically. This happens when the transformation does not preserve the semantics of the application (e.g., in EGAP #6 when the constructor of a HashMap receives a Map object, it is not possible to change it into an ArrayMap, as the former has no such constructor).

We were able to detect at least one EGAP in **239** applications, and overall we detected **71** different combinations of EGAPs. Out of these 71, we were able to refactor, at least in one application, **44** combinations, which then consist of our (EGAP combination) study base. Overall, we obtained **416** different applications where at least one refactoring was applied, which consist of our (application) study base[10].

This represents a considerable larger study base than reported in the literature.

4.4.3 Results and Discussion

Table 4.2 summarizes the results of the EGAP detection and refactoring phases. The first column contains the total set of combinations that were found in at least one application, and the second column shows the number of applications on which each combination was detected. Columns

[9] AutoRefactor webpage: autorefactor.org (visited on October 19, 2020).

[10] Note that the same (original) application is being counted more than once, in case more than one combination of EGAPs was actually applied to it.

Max per app. and **Min per app.** respectively show the highest and lowest number of times that EGAPs in the combination were detected in an application. For example, for combination [#1, #6], the application in which the highest number of (#1+#6) EGAPs were found totaled 5 occurrences (4 of EGAP #1 and 1 of EGAP #6). Finally, column **# Apps Refactored (%)** shows the success rate in refactoring that combination[11].

From the set of 609 analyzed applications, we observed that:

1. EGAPs #9, #10 and #11 were not found in any application. Despite their relevance described in the literature [Vekris et al., 2012, Jiang et al., 2017], this suggests that the contexts where they were found might not be generally representative;

2. on **370** of them ($\approx 60\%$) none of the 11 EGAPs were detected. This suggests that there still might be room to propose new energy greedy source code patterns or that potentially a significant amount of application developers already follow the best practices for energy efficient code;[12]

3. EGAPs #6, #7, and #8 are the most frequent, either individually or combined. This might be related to the fact that these three particular EGAPs are not exclusively related with the Android API. Indeed, any Java application can use an HashMap, and this seems to confirm that not declaring a method as static when it is possible to do so, or unnecessarily calling a method inside a loop (e.g., to improve readability), is common. Nonetheless, it is important to notice that not only Google itself suggests to avoid using the HashMap collection for performance reasons, but it was also already proved that this collection can not only be energy-greedy in Android applications, but also in typical Java applications [Pereira et al., 2016, Hasan et al., 2016].

4.5 Analyzing The Energy Impact Of Refactoring EGAPs

The goal of our study is to understand the energy impact of refactoring EGAPs. In this section, we explain the procedure we followed: the tools and methodology used, the data collected, and how it was used to reason about energy impact.

[11] Combinations with a success rate of 0% are available in the supplementary material and were omitted due to space limitations.

[12] In the latter case, note that developers may already request EGAPs #1 - #5 to be automatically signaled upon the application build.

Table 4.2: Results Of EGAP Detection And Refactor Phases

EGAP Combination	#Apps Detected	Max per app.	Min per app.	# Refactored apps (%)
[#1]	12	39	1	5 (41.7%)
[#2]	2	1	1	1 (50%)
[#3]	29	11	1	5 (17.2%)
[#4]	40	39	1	39 (97.5%)
[#5]	5	6	1	4 (80%)
[#6]	84	21	1	60 (71.4%)
[#7]	111	53	1	50 (45%)
[#8]	172	155	1	101 (58.7%)
[#1, #6]	5	[4, 1]	[1, 1]	2 (40%)
[#1, #7]	5	[1, 8]	[1, 1]	2 (40%)
[#1, #8]	7	[39, 5]	[2, 2]	1 (14.2%)
[#3, #4]	5	[5, 3]	[1, 1]	2 (40%)
[#3, #6]	17	[7, 21]	[1, 1]	2 (11.8%)
[#3, #8]	20	[7, 155]	[1, 1]	1 (5%)
[#4, #5]	2	[4, 1]	[1, 2]	2 (100%)
[#4, #6]	17	[1, 20]	[1, 1]	11 (64.7%)
[#4, #7]	20	[39, 4]	[1, 1]	10 (50%)
[#4, #8]	26	[2, 30]	[1, 1]	7 (26.9%)
[#5, #6]	3	[6, 4]	[2, 1]	2 (66.6%)
[#5, #7]	3	[1, 8]	[1, 1]	2 (66.6%)
[#5, #8]	3	[6, 8]	[2, 6]	1 (33.3%)
[#6, #7]	47	[21, 53]	[1, 1]	12 (25.5%)
[#6, #8]	62	[21, 155]	[1, 1]	23 (37.1%)
[#7, #8]	76	[53, 155]	[1, 1]	25 (32.8%)
[#1, #6, #7]	3	[3, 1, 5]	[1, 1, 1]	1 (33.3%)
[#1, #6, #8]	4	[3, 1, 18]	[1, 1, 3]	1 (25%)
[#1, #7, #8]	5	[3, 5, 18]	[1, 1, 6]	2 (40%)
[#3, #4, #6]	4	[1, 1, 20]	[5, 3, 2]	2 (50%)
[#4, #5, #6]	1	[4, 1, 6]	[4, 1, 6]	1 (100%)
[#4, #5, #7]	2	[4, 1, 8]	[1, 2, 4]	1 (50%)
[#4, #5, #8]	1	[4, 1, 10]	[4, 1, 10]	1 (100%)
[#4, #6, #7]	13	[1, 20, 17]	[1, 1, 1]	5 (38.5%)
[#4, #6, #8]	16	[1, 20, 18]	[1, 1, 2]	7 (43.8%)
[#4, #7, #8]	16	[2, 12, 30]	[1, 1, 2]	5 (31.2%)
[#5, #6, #7]	1	[1, 6, 8]	[1, 6, 8]	1 (100%)
[#5, #6, #8]	3	[6, 4, 8]	[2, 1, 6]	2 (66.6%)
[#5, #7, #8]	1	[1, 8, 10]	[1, 8, 10]	1 (100%)
[#6, #7, #8]	40	[21, 53, 155]	[1, 1, 1]	8 (20%)
[#1, #6, #7, #8]	3	[3, 1, 5, 18]	[1, 1, 1, 6]	1 (33.3%)
[#4, #5, #6, #7]	1	[4, 1, 6, 8]	[4, 1, 6, 8]	1 (100%)
[#4, #5, #6, #8]	1	[4, 1, 6, 10]	[4, 1, 6, 10]	1 (100%)
[#4, #5, #7, #8]	1	[4, 1, 8, 10]	[4, 1, 8, 10]	1 (100%)
[#4, #6, #7, #8]	12	[1, 20, 17, 18]	[1, 1, 1, 2]	3 (25%)
[#4, #5, #6, #7, #8]	1	[4, 1, 6, 8, 10]	[4, 1, 6, 8, 10]	1 (100%)

4.5.1 Experimental Setup

Our experiments were performed in three factory-reseted Nexus 5 mobile devices running Android 6.0.1. Energy consumption data was obtained using the (same version of the) Qualcomm Trepn

profiler[13], a profiling application that has been used in several energy-related studies [Jabbarvand et al., 2015, Hegde et al., 2016, Hu et al., 2018].

We divided the workload through all devices in a way that all applications to be tested for each combination of EGAPs (both before and after refactoring) are executed in the same device. This ensures not only that the comparison between the different versions of the same application, but also the analysis of the impact of one EGAP combination, are made using data gathered from the same device.

During each test execution, the only applications running were, aside from OS-related Android applications and services, the application under test and Trepn (running as a service, in background). Moreover, we kept all devices in airplane mode and with minimum screen brightness, to reduce as much as possible the measurement noise.

The EGAPs we considered can be found in different components in an Android application (e.g., the `Wakelock` is associated only with Activities, while the `HashMap Usage` can occur anywhere in the application). Thus, we decided to test the applications where the EGAPs were transformed by running different usage scenarios on them.

4.5.2 Experimental Pipeline

For the purpose of testing the energy effects of refactoring an EGAP, we developed a testing framework called Chimera[14], which incorporates EGAP detection and refactoring. This framework is also capable of automatically running a predefined set of usage scenarios, storing the results of each scenario, and grouping the results per application and EGAP. A more detailed explanation of this framework is included in Section 6.5.

Chimera implements an execution pipeline, which is depicted in Figure 4.1. Essentially, the input to this pipeline are the list of EGAP combinations to analyze, and the path to the repository of Android applications. A preliminary step is performed on the repository, where the Detector operator reports all existing EGAPs within each application, and stores this information to be later used. The initial step in the pipeline consists of, for each EGAP (or combination) C, using the Filter operator to select all applications where all EGAPs in C where detected. This uses the

[13]Trepn webpage: developer.qualcomm.com/software/trepn-power-profiler (visited on October 19, 2020).
[14]Chimera source code: https://gitlab.com/MarcoCouto/android-egaps (visited on October 19, 2020).

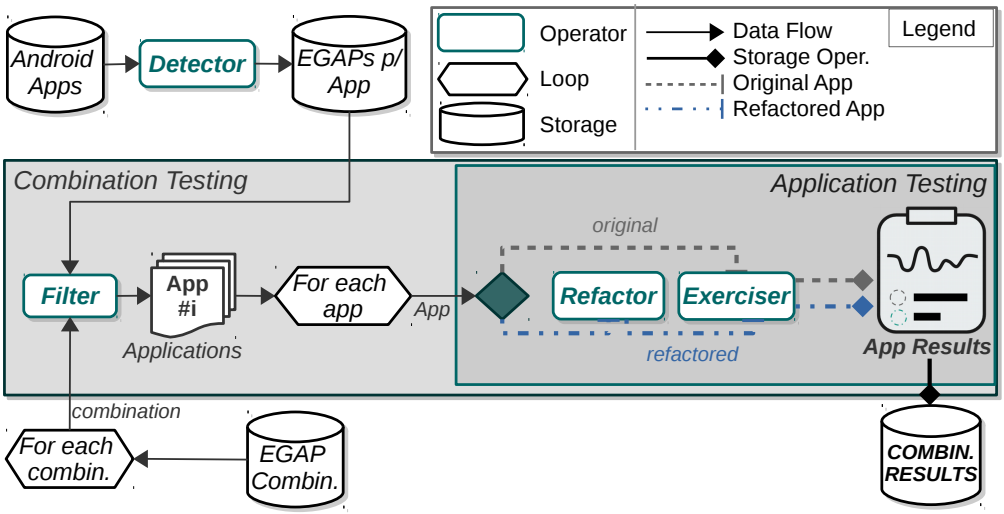


Figure 4.1: Case Study Pipeline

reports from the Detector operator.

The list of matching applications is passed to the second step of the pipeline. Here, each application goes first trough the Refactor operator, which creates two versions of it: a plain copy, and a version with the EGAPs refactored. These two versions are passed to the Exerciser operator, which is responsible for automatically running the exact same set of tests/usage scenarios on both versions. It also records relevant data from each test execution, such as energy consumed and elapsed time. These results are then stored and indexed by application and EGAP/combination.

4.5.3 Results and Discussion

The experiments described in the previous section took over **500 hours** to finish. Once finished, we obtained results for **416** applications. For each application, and for each of its 25 test scenarios, the experiment produced i) the number of times each EGAP was refactored and ii) the energy consumed by the original (E_O) and refactored (E_T) versions. Using this information, we were able to calculate the **gain** obtained in each test, which is the percentage difference between E_O and E_R, calculated using the formula: $\frac{(E_O - E_R)}{E_O} \times 100$. A negative value means a **loss** of energy in a test.

The results of the experiments are summarized in Figure 4.2[15]. For each EGAP (or combination), we included a bar plot comparing the proportion of usage scenarios which resulted in

[15]We also include in the appendix, from Figure B.1 to B.44, the individual results for each EGAP or combination.



Figure 4.2: Overview: Experimental Results

gains with the ones that resulted in losses, with the respective number in front of each bar. The boxplots indicate how the exact gains/losses values are distributed[16]. Combining these two plots, we can observe not only if a certain combination had overall more gains/losses, but also if the impact was higher.

We can see that, when refactoring individual patterns, the number of tests that resulted in gains is always greater than the number of losses. Moreover, as the boxplots confirm, the gain values are always considerably higher than the loss values. In line with the literature, this confirms that refactoring such patterns individually consistently leads to energy savings.

Looking at combinations of EGAPs, the majority of them do clearly have higher gains than

[16]Outliers were excluded for the purpose of aiding data visualization.

losses. Nevertheless, in a few cases the gains do not clearly surpass the losses, or are even lower. For instance, combination [#4, #5] has the exact same number of tests with gains and losses, and the boxplots are very similar, whereas [#4, #8] has more tests which resulted in gains, but the boxplots indicate a slight tendency to favor the losses. The more interesting scenario, however, occurs for combination [#4, #5, #6, #7]: the distribution of gain values demonstrates that they were considerably low (all below ≈3%), and are clearly surpassed by the losses.

Based on our results, we can safely state that **when refactoring patterns individually, the gains are consistently higher than the losses, but the same cannot be concluded for a few combinations of patterns**, hence answering **RQ1**.

In order to answer **RQ2**, we examined, for each individual refactoring, how similar were its gains/losses when compared to the other 7. We ran a Mann–Whitney U test, which is used to assess if two independent samples (in our case, a set of gains/losses) are selected from populations having the same distribution. If H_0, the null hypothesis, is rejected, the gains/losses of two refactorings are significantly different. Also, the resulting test statistics can assess if a value from one sample is likely to be higher or lower than one randomly taken from the other, and by how much. We consider $\alpha = 0.05$, so we can reject the null hypothesis with 95% confidence.

Table 4.3: Results of Mann–Whitney U test for individual combinations

	#1		**#2**		**#3**		**#4**		**#5**		**#6**		**#7**		**#8**	
	ρ	stats	ρ	stats	ρ	stats	ρ	stats	ρ	stats	ρ	stats	ρ	stats	ρ	stats
#1	–	–	0	-7.19	0	-5.64	**0.98**	0.03	0	-2.71	**0.41**	-0.83	**0.75**	-0.32	**0.69**	-0.41
#2	0	7.19	–	–	0	5.49	0	7.46	0	6.38	0	7.66	0	7.14	0	7.40
#3	0	5.64	0	-5.49	–	–	0	7.19	0	2.82	0	6.86	0	6.71	0	6.91
#4	**0.98**	-0.03	0	-7.46	0	-7.19	–	–	0	-3.32	**0.05**	-1.97	**0.39**	-0.86	**0.34**	-0.96
#5	0	2.71	0	-6.38	0	-2.82	0	3.32	–	–	0	2.76	0	2.92	0	2.96
#6	**0.41**	0.83	0	-7.66	0	-6.86	**0.05**	1.97	0	-2.76	–	–	**0.28**	1.09	**0.23**	1.20
#7	**0.75**	0.32	0	-7.14	0	-6.70	**0.39**	0.86	0	-2.92	**0.28**	-1.09	–	–	**0.98**	0.03
#8	**0.69**	0.41	0	-7.40	0	-6.91	**0.34**	0.96	0	-2.96	**0.23**	-1.20	**0.98**	-0.03	–	–

The test results are depicted in Table 4.3. Each row represents an EGAP, and it contains the p-value (ρ) and test statistics ($stats$) resulting from testing the gains/losses of that EGAP against the ones of the EGAP in the corresponding column (p-values below 0.01 were rounded to 0).

From the obtained p-values, most gains are expected to be quite different between refactorings (rejected H_0). In a few cases, a high p-value is observed; here, the expected gains are very similar for both refactorings. We see that, for instance, `DrawAllocation` (#1) is expected to produce

similar gains as `ObsoleteLayoutParam` (#4) and `ExcessiveMethodCalls` (#7) (p-values of 0.98 and 0.75).

When H_0 is rejected, the $stats$ value gives us an indication on what to expect from the results of two refactorings. By comparing, for instance, `Recycle` (#3) and `DrawAllocation` (#1), the stats value is 5.64, and when comparing with `Wakelock` (#2) the value is -5.49. This means that the average gain from refactoring `Recycle` is expected to be about 5.64 times bigger than the ones obtained from refactoring `DrawAllocation`, but at the same time about 5.49 times lower when compared with `Wakelock`.

In conclusion, to answer **RQ2**, **refactoring individual EGAPS most often, but not always, leads to energy savings with significantly different orders of magnitude**.

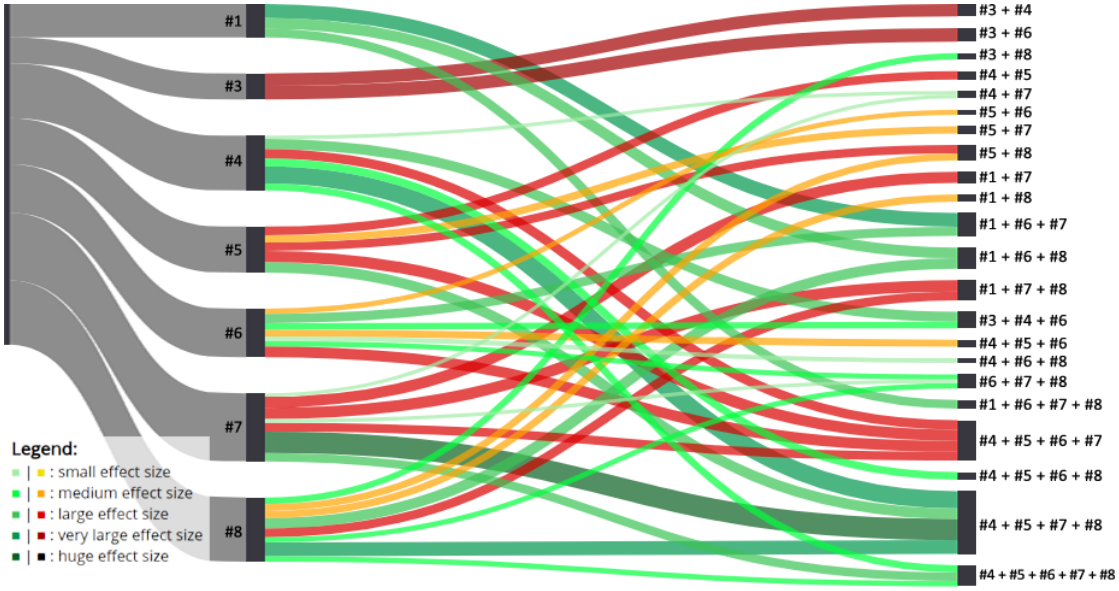


Figure 4.3: Gains Comparison: EGAPs vs Combinations

With **RQ3**, we first sought to understand whether it is always preferable to apply, in an application, all the refactorings that are possible. For this, we compared every single refactoring, with all the combinations where such refactoring is included. For fairness, we considered only the applications where both the individual refactoring and the combinations which include it are available.

Again, a Mann–Whitney U test was used to compare the gains of every individual refactoring R_i against all combinations C_i where it occurs[17]. If H_0 is rejected, there is statistical evidence that the gains obtained from applying R_i are either consistently higher or lower than the ones

[17]The results of these tests are included in appendix, from Table B.1 to B.7.

obtained from applying C_i. If so, we also wanted to quantify how big that difference is. We calculated the Cohen's d value between gains obtained from R_i and C_i, and we used the Sawilowsky [Sawilowsky, 2009] suggested thresholds of 0.2, 0.5, 0.8, 1.2, and 2, which respectively translates to small, medium, large, very large and huge effect size.

The results we obtained are depicted in Figure 4.3[18]. Each individual refactoring on the left is connected to the combinations on the right on which the statistical test suggested to reject H_0 (i.e. where the gains between them are undeniably different, when applied on the same applications). A green connection reveals a tendency for the gains to be higher for the combination on the right; we shall refer to such scenarios as positive effects. Similarly, yellow, orange, or red connections indicate scenarios where the gains tend to be lower, and we shall call them negative effects. The color intensity and the connection width express the effect size.

If it would be preferable to apply as much refactorings as possible, then all connections should express positive effects, or at least the negative ones should not exist (which would ultimately mean that there was no strong evidence). In practice, out of the existing 38 connections, 22 are positive and 16 are negative, with the latter having an overall more relevant effect size. In conclusion, and to provide the answer to **RQ3**, we can say that **combinations that contain positive connections are the ones with higher gains**, as such connections reflect the existence of higher and consistent gains. Still, the existence of negative connections provides evidence that **the gains obtained by some individual refactorings can be significantly reduced when combined with others**.

We now seek to provide an answer to **RQ4**. By doing so, we aim at providing a guideline for developers to follow when aiming at reducing energy consumption of an application, by refactoring energy greedy patterns. In other words, we want to identify which combinations should be avoided/adopted, and how to decide on the potentially ambiguous cases.

We believe our recommendations should be interpreted as a confirmation of the possibility that, for certain combinations of refactorings, the energy savings are not cumulative. We argue that these results present a way of looking at the **most likely** scenario for a combination, when it comes to energy savings.

For the majority of combinations, the results indicate that the gains are consistently higher

[18] An interactive version of this plot is available in the online appendix.

than the losses (as shown in Figure 4.2). However, for the ones identified in Figure 4.3 with having negative connections, applying only one refactoring in the same context has proven to result in higher energy savings. A joint and more in-depth analysis of both figures allows us to propose the following guidelines:

$\mathscr{G}_1$**: Avoid combinations whose gains are not significantly larger than the losses** (Figure 4.2), **specially when they have at least one negative connection** (Figure 4.3). Looking, for instance, at combination [#4, #5, #6, #7], we see that it has 4 negative connections, one for each of its individual refactorings. Additionally, it had more tests resulting in losses, and with a higher impact compared to the the ones resulting in gains. Another example is the combination [#5, #8]. We consider these combinations to be **harmful**.

$\mathscr{G}_2$**: Adopt combinations that (only) have positive connections.** This is the case of 11 combinations, that we argue developers should apply. Not only do their gains surpass their losses, but there is also statistical evidence that the gains are expected to increase if all refactorings in the combination are applied, instead of applying only one of them. Combination [#4, #5, #7, #8] is the one where this is more notorious: all tests resulted in gains, and the combination has 4 positive effect connections, with either a large, very large, or huge effect size. We classify these combinations as **effective**.

$\mathscr{G}_3$**: Even when considering a combination that resulted in substantially higher gains, it might be preferable to do individual refactorings if the combination (only) has negative connections or no connections at all.** Combinations such as [#3, #4] resulted in substantially higher gains but have negative connections. This means that, while the combination is **helpful**, further analysis is needed to decide on what to refactor; this may involve, e.g., counting the occurrences of EGAPs #3 and #4 in a particular application.

4.6 Threats To Validity

We focused on studying the impact of refactoring EGAPs over a large set of applications and on scenarios that try to simulate realistic usage. There are several aspects that may affect the validity of our work and findings.

1. While we have used an experimental setup that is not totally aligned with related works, the fact

is that such works have themselves implemented different setups. Also, although some potential measurement noise may exist due to unpredictable actions, such as activity management or garbage collection, there are strong evidences that our conclusions are drawn upon correct data. For once, our results for individual EGAPs (the ones for which a comparison is possible) are well aligned with the findings in the literature. Also, running 25 scenarios per application aids in minimizing the potential error margin. Finally, Trepn is accepted as an effective measurement tool, as it was already used in multiple other energy-related studies.

2. Using random events to define tests may not ensure that applications have all their features explored. However, this is the only generic way to simulate realistic, context-free, usage scenarios, which are essential within our methodology.

3. Our conclusions are sustained on the gains/losses obtained for each refactoring or combination. Extensive as our work may be, this might not reflect all possible scenarios Nevertheless, since our methodology ensures that any significant gain/loss is due to the performed refactorings only, and we test a very large number of applications 25 times without enforcing the use of refactored code, we argue that our study provides a compelling evidence for the most likely scenarios.

4.7 Conclusions

In this chapter, we detailed the aspects of a study where we performed an analysis over a large-scale repository of Android applications, using as subjects, individually and combined, 11 code patterns termed EGAPs, which were individually studied in previous research to understand how replacing them can influence energy consumption. Our analysis considered the global impact that refactoring has on an application, which also differs from previous approaches.

Our work provides several findings that can guide developers in improving the energy efficiency of their code. For once, refactoring individual EGAPs consistently leads to energy savings, which is aligned with previous findings. Refactoring combinations of EGAPs, however, can sometimes produce less gains than expected, so the decision on whether or not to refactor a certain combination needs to be properly evaluated. The analysis is fully automated, and was achieved by open-source tools, which are publicly available for others to use.

We believe our work raises a relevant direction for future research. Although our extensive and in-depth study has provided experimental and statistical evidence that combining as much refactorings as possible may sometimes not produce the optimal energy efficiency, it would be interesting to further understand, when it occurs, why this is the case. Nevertheless, we strongly believe this direction should be addressed by a dedicated study, that can build on our (novel) findings.

5 Towards Combining Energy Smells with Decision Support

This chapter introduces the concept of energy debt: a new metric, reflecting the implied cost in terms of energy consumption over time, of choosing a flawed implementation of a software system rather than a more robust, yet possibly time consuming, approach. A flawed implementation is considered to contain code smells, known to have a negative influence on the energy consumption.

Similar to technical debt, if energy debt is not properly addressed, it can accumulate an energy "interest". This interest will keep increasing as new versions of the software are released, and eventually reach a point where the interest will be higher than the initial energy debt. Addressing the issues/smells at such a point can remove energy debt, at the cost of having already consumed a significant amount of energy which can translate into high costs.

We present all underlying concepts of energy debt, bridging the connection with the existing concept of technical debt. We describe our approach with a preliminary motivational example, showing how to compute the energy debt of a real-world application. A prototype is under development, which already includes the detection of several Android energy smells, and calculates the energy debt and energy interest of Android applications.

5.1 Introduction

Technical Debt (TD) describes the gap between the current state of a software system and the ideal state of that same software. The key idea of technical debt is that software systems may include artifacts which can be hard to understand/maintain/evolve, causing higher costs in the future software development and maintenance activities. These extra costs can be seen as a type of debt that developers owe the software system.

Although technical debt is still a recent area of research, it has gained significant attention over the past years: A recent systematic mapping study [Li et al., 2015] identified ten different types of technical debt, namely requirements, architectural, design, code, test, build, documentation, infrastructure, versioning, and defects technical debt. In fact, TD is a concern both for researchers and software developers. The current widespread use of non-wired computing devices is also making energy consumption a key aspect not only for hardware manufacturers, but also for researchers and software developers [Pinto and Castor, 2017]. Indeed, several energy inefficient programming practices have been reported in literature, namely, energy patterns for mobile applications [Cruz and Abreu, 2017, Couto et al., 2020a], the energy impact of code smells [Palomba et al., 2019, Morales et al., 2018, Saborido et al., 2018], energy-greedy API usage patterns [Linares-Vásquez et al., 2014], energy (inefficient) data structures [Pereira et al., 2016], programming languages [Pereira et al., 2017b], etc. which do have significant impact on the energy consumption of software.

All these research works show that energy-greedy programming practices, also called energy smells, do often occur in software systems. These can be attributed to the current lack of knowledge software developers have in order to build energy efficient software, and the lack of supporting tools [Pinto and Castor, 2017].

This chapter defines energy debt as the additional estimated energy cost of executing a software system, due to the occurrence of energy smells in the software's source code, when compared to the estimated energy cost of executing the non-energy smelly (i.e. energy ideal) version of that same software. To express energy debt we consider a set of energy code smells presented in the current state of the art literature on green software, together with the energy savings reported in the studies where such smells have been presented. Thus, the energy debt of a program is computed after knowing the number of occurrences and their locations in the program's source

code: energy smells inside loops/recursion, single statements, or inside dead code do have different debt weights.

This sections within this chapter are structured as follows: Section 5.2 thoroughly describes the notion of our novel concept of energy debt, and how it should be expressed/calculated; in Section 5.3 we present a preliminary experiment where our technique is applied in a real-world application; finally, our conclusions and future work are included in Section 5.4.

5.2 Introducing Energy Debt Concepts

In this section, we will explain a novel concept called energy debt, which is very much aligned with technical debt in the sense that it presents developers and decision makers with information regarding the evolution of energy inefficiency of their software systems. As presented in a recent ACM communications [Pinto and Castor, 2017], developers fall into energy-greedy practices and tendencies due to the lack of knowledge and the lack of tools to help understand, locate, and optimize energy inefficiencies. Additionally, other practicioners and decision makers also lack the necessary tools to help interpret how energy inefficiencies can impact their product lifecycle, and what they should focus on tackling in order to reduce energy costs [Manotas et al., 2016].

5.2.1 Concept Overview

Before we present the definition of energy debt, let us recall the metaphor of technical debt. Technical debt reflects the cost arising from performing additional work on a software system, due to developers taking "shortcuts that fall short of best practices" [Allman, 2012]. Hence, this cost can be defined as the technical effort, in working hours, required for fixing all issues associated with bad programming practices, in a given release. The cost keeps increasing, as new versions (with new issues) keep getting released, and if the initial issues are not properly addressed, they accumulate interest [Chatzigeorgiou et al., 2015].

Based on the underlying concept of technical debt, we define **Energy Debt** as the amount of unnecessary energy that a software system uses over time, due to maintaining energy code smells for sustained periods.

A visual comparison of the two concepts is depicted in Figure 5.1. The left-hand side of

the figure illustrates the well-known representation of technical debt, including the concepts of refactoring and maintenance effort, along with the definition of interest. On the right-hand side we present the definition of energy debt, where we assume that evolving the software (i.e., introducing new features on new releases) will eventually result in the addition of new (energy) code smells, hence the Energy Debt (ED) increases per version.

The main difference between technical and energy debt, at this point, is the fact that the former can be presented as a unique cost value expressing how much effort would be necessary to address the issues, whereas the same approach cannot be applied to the latter. The cost of maintaining energy code smells in a software release is always directly proportional to the amount of time that the same release operates. As an example, if two software systems S_1 and S_2 have the exact same energy code smells, the amount of excessive energy consumed by S_1 might be higher than S_2 if it is intended to be used longer, during the same timespan.

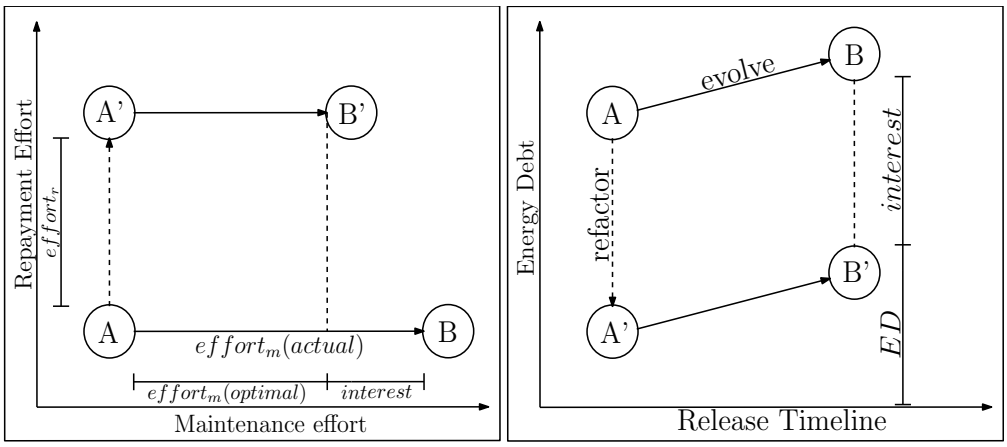


Figure 5.1: Technical Debt vs Energy Debt Terminology

Given the previous assumptions, we argue that the energy debt ED of a software release must be expressed not as a cost value, but as a cost function, which receives, as input, two variables: a software release r, and a usage time t. Equation 5.1 defines such a function, and it allows us to obtain, for a given release r, its energy debt ED after a given usage time of t:

$$ed(r,t) = cost(r) * t \tag{5.1}$$

The $cost(r)$ function included in the equation represents the energy cost of release r, per unit of time. In other words, it relates to the existing number of energy code smells in that version, and the energy cost (per unit of time) of each one. The definition of that function is expressed as

Equation 5.2:

$$cost(r) = \sum_{i=1}^{N} w_i(r) \times E(i) \tag{5.2}$$

Here, N is the number of smells included in the considered catalog, while $w_i(r)$ returns a weight value for smell i, which is affected by the number of i smells found in release r and the context in which they were found (we will discuss this with greater detail in Section 5.2.2). $E(i)$ returns the expected energy debt per time unit of smell i, as defined in the smell catalog.

The formulas presented thus far assume that each considered energy code smell has an associated energy debt value, expressed in function of time units (for instance, per minute). Nevertheless, when studying the energy consumption impact of code smells, researchers often tend to present the potential gains/savings as an interval (i.e., highest and lowest observed energy saving).

The highest/lowest saving approach adds valuable information regarding potential energy savings. Fixing a certain smell can result in savings between, e.g., 150mJ and 3000 mJ per minute. When compared to another smell with savings between 300 mJ and 900 mJ per minute, we know that in a best-case scenario refactoring the first one would result in higher gains, but in a worst-case scenario the second presents better savings. Hence, a developer can use this information to decide how to properly focus their attention when refactoring code smells, depending on the project goals [Couto et al., 2020a].

In accordance with the previous assumption, we decided that our approach for energy debt should consider, for each code smell, two energy values: the highest (E_{max}) and lowest (E_{min}) observed energy savings. Since energy debt must be expressed as a function of the usage time, it is expected that E_{max} will be much higher with the increase of usage time, as depicted in Figure 5.2.

There are two represented versions of a release in this figure: the optimal version, with all smells removed (A'), and the energy smelly version (A). The optimal version already has a constantly increasing energy consumption, as it would be expected. Energy debt can be summed up as the area between the line for A', and the (red) line for A, which becomes much larger when considering the maximum values. This will introduce changes to Equation 5.1, which will consider two cost values, in the form of two functions:

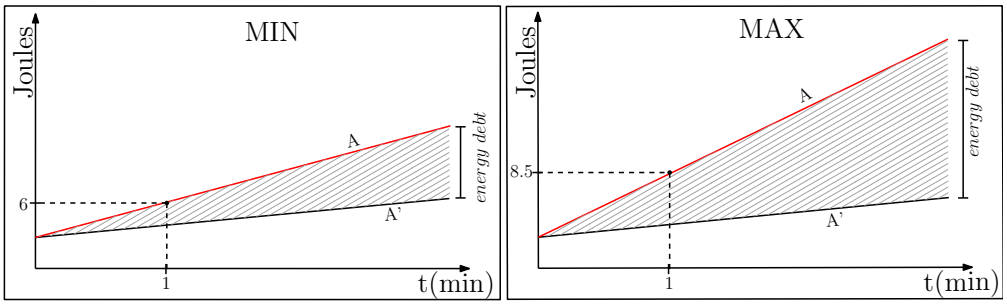


Figure 5.2: Energy Debt Thresholds Increase Over Time

$$ed(r,t) = \left(cost_{min}(r) \times t; cost_{max}(r) \times t\right) \tag{5.3}$$

The energy debt will therefore always be presented in the form of an interval. Consequently, each of the cost functions will need to refer to the proper energy debt per time unit. In other words, the $E(i)$ function in Equation 5.2 will be $E_{min}(i)$ for the lowest savings, and $E_{max}(i)$ for the highest savings.

The final requirement is, as previously stated, an energy smell catalog, which should map lower and upper-bounded energy estimations to each considered smell, These estimations need to be dependent of the elapsed time, i.e., they will be (as described above) functions that multiply the provided time value by a consumption constant. A possible and valid approach to achieve this is to build the catalog from the results obtained in previous studies, such as the ones we referenced in Section 4.2. Hence, by examining the reports of those studies, we can extract this information, and associate to each pattern an approximation to their maximum and minimum cost (i.e. potential savings) per time unit. The results of this task (i.e., the energy debt supported catalog) is depicted in Table 5.1.

5.2.2 Counting Expenses & Estimating Debt

The next step towards estimating energy debt is to define a strategy to analyze the occurrence of such smells in a given release. The starting point for this task will be to use a common source code analysis tool capable of detecting code smells. There are several ways to achieve this. For instance, SonarQube[1], which is a widely used tool for technical debt estimation, provides an API

[1]SonarQube webpage: http://www.sonarqube.org (visited on October 19, 2020).

Table 5.1: Energy Debt Smell Catalog

	Abbrev.	E_{min} (mJ per min)	E_{max} (mJ per min)
`DrawAllocation`	**DA**	36	158
`Wakelock`	**WL**	10	194
`Recycle`	**R**	15	533
`ObsoleteLayoutParam`	**OLP**	94	561
`ViewHolder`	**VH**	892	2105
`HashMaUsage`	**HMU**	28	229
`ExcessiveMethodCalls`	**EMC**	557	9529
`MemberIgnoringMethod`	**MIM**	89	7844

for defining detection rules for issues/smells of different languages.[2]

Detecting smell occurrences, however, is a necessary but not the sole requirement to properly analyze its impact on energy debt. A smell can be detected, for instance, inside a block of dead/unreachable code, or it can be placed inside a procedure which may only be executed once in a software lifecycle (eg. an initial setup). On the other hand, a code smell can also be part of a mechanism designed to be re-utilized several times, such as a loop or a thread. These scenarios should be considered when estimating energy debt, and since our approach implies using statical analysis mechanisms, we can follow already defined strategies for static energy analysis.

A very common and well-established approach for these situations is to define weights for smells, depending on the context on which they occur. For instance, Jabbarvand et al. [Jabbarvand et al., 2015] defined a strategy for weighing instructions which might be repeated. First, it extracts the full method call graph of a program, and provide for each method an energy score; such a score depends on 3 things: (i) how many paths can be taken to reach that node from the root node, (ii) whether it is found inside a loop, and if so (iii) what is the expected loop's bound; this statically obtained information is then used to increase/decrease the energy score of the node.

Several strategies have been suggested for this task, all of which accepted by the community. This leads us to believe that, although it is important to weight code smells depending on the occurrence context, several factors can influence the decision on what approach to follow (e.g. how much information is extracted from the smell detection tool, or trading off information detail

[2] Adding Coding Rules webpage: https://docs.sonarqube.org/latest/extend/adding-coding-rules/ (visited on October 19, 2020)

with the analysis time). Hence, we argue that the selected strategy is also context dependent, and can be as simple or as detailed as desired. Nevertheless, whatever approach one follows, an update to Equation 5.2 is necessary to consider it. As an example, we considered a simplification of the strategy from Jabbarvand et al. [Jabbarvand et al., 2015]:

$$w(i,r) = \sum_{j=1}^{C} paths(j) \times LB \tag{5.4}$$

In this equation:

- C is the number of i smells found in the release r;
- $paths(j)$ represents the number of paths in the call graph through which the j^{th} occurrence of smell i is reachable;
- LB will be 1 **if** the j^{th} of the smell is outside a loop, or a constant indicating the loop bound; it can be inferred if possible, or pre-established.

In order to better explain how all these concepts connect with each other, when aiming at estimating the energy debt of different software releases, we have prepared a running example, depicted in Figure 5.3. In this example, we have a catalog with 3 smells, each one with the energy gains thresholds defined (values are in milliJoules per minute), and 3 releases with the analysis report for each. The report is a list of the detected smells, where for each one there is information regarding (i) the number of paths through which the smell is reachable ($paths$), and (ii) whether it was found inside a loop ($LB > 1$) or not ($LB = 1$).

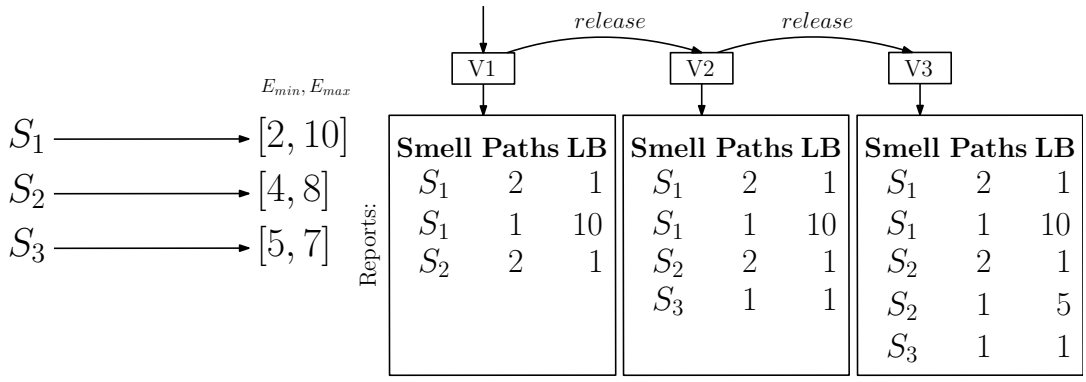


Figure 5.3: Estimating Energy Debt per Release

Using the formula from Equation 5.4, we can determine the weight to be applied to each smell. For example, for release $v1$, the weights for smell $s1$ and $s2$ would be:

$$w(s1, v1) = (2 \times 1) + (1 \times 10) = 12$$

$$w(s2, v1) = (2 \times 1) = 2$$

We can apply the computed weights in the formula from Equation 5.2, to obtain an estimated value for the energy debt of release $v1$. As previously mentioned, our energy debt definition considers two reference values: the lowest and highest estimated energy debt. This means that the $cost$ function in Equation 5.2 must be computed twice: the first using the lowest estimated gains per smell (E_{min}), and the second using the highest (E_{max}). Once again, for release $v1$, we would have the following $cost_{min}$ and $cost_{max}$ values:

$$cost_{min}(v1) = (w(s1, v1) \times E_{min}(s1)) + (w(s2, v1) \times E_{min}(s2))$$

$$\Leftrightarrow cost_{min}(v1) = (12 \times 2) + (2 \times 4) = 32$$

$$cost_{max}(v1) = (w(s1, v1) \times E_{max}(s1)) + (w(s2, v1) \times E_{max}(s2))$$

$$\Leftrightarrow cost_{max}(v1) = (12 \times 10) + (2 \times 8) = 136$$

These two reference values represent the energy debt for release $v1$. This means that energy debt can vary from a minimum of 32 to a maximum of 136 milliJoules per minute. As explained previously, energy debt is expressed as a function of usage time. Therefore, if one wants to know how much debt this release accumulates after being used for, e.g., one hour, this can be estimated as follows:

$$ed(v1, 60min) = \left(cost_{min}(v1) \times 60; cost_{max}(v1) \times 60\right)$$

$$\Leftrightarrow ed(v1, 60min) = \left(1,920mJ; 8,160mJ\right)$$

The estimated values indicate an energy debt varying between 1.92 and 8.16 Joules per hour. This means that, for every hour that release $v1$ is being used, it could be consuming **at least** $1.92J$ less, and the savings could be **up to** $8.16J$.

Finally, it is important to point out that the accuracy of the estimated thresholds rely on the adequacy/robustness of the analysis components, namely the smells catalog, the code analysis tool, and the weighing function for repeated smell's executions. It is possible to use our energy debt approach to compute reference values for the energy inefficiency of a release, rather then to produce extremely accurate estimates of the potential energy savings per usage time. It depends on how one wants to apply the concept.

5.2.3 Paying Interests

The concept of interest in technical debt has already been formulated [Chatzigeorgiou et al., 2015], and its practical application has also been studied [Ampatzoglou et al., 2018, Tsintzira et al., 2019, Ampatzoglou et al., 2015]. The concept is based on the fact that, as a software system evolves (i.e., new versions are released), the cost/effort of adding features to a new release (maintenance effort, expressed as working hours) keeps increasing if the task of addressing the technical debt keeps being postponed. Maintaining a release with technical debt requires more effort than to maintain the same release without it; the effort difference between the two is called the technical debt interest.

The left-hand side of Figure 5.1 illustrates the interest concept. There is a software version, A, containing code smells, and therefore technical debt. At this point, a decision can be made on what to prioritize: (i) invest effort in fixing the smells (repayment effort) and release an optimal version of that release, A', without technical debt, or (ii) release the version with the smells. If the priority is (ii), then the evolution effort to a new release B will be higher. This additional effort could be avoided, but the priority was releasing a new version, which can happen for a wide variety of reasons (e.g. client demands, faster market reach, etc.); this resembles the idea of accumulation of debt, and debt needs to be re-payed.

Chatzigeorgiou et al. [Chatzigeorgiou et al., 2015] presented a technique to predict the technical debt breaking point, i.e., when the accumulated interest is higher than the initial effort to remove the technical debt (i.e., the principal). With this, it is possible to present developers another choice: if technical debt keeps being "ignored", then they have approximately until release number N to properly deal with it; otherwise, from that moment on, the additional maintenance effort will always be higher than the effort to deal with the principal. In conclusion, at that point

they are wasting development time.

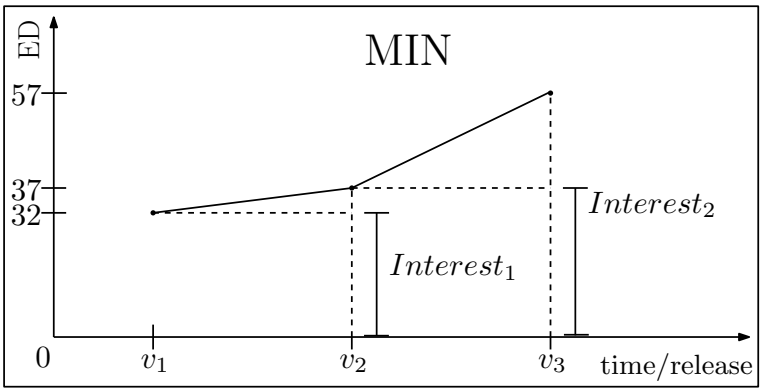


Figure 5.4: Accumulation of Interest

When considering energy instead of technical debt, the interest concept needs another definition. First, it will not indicate how much more maintenance effort is being applied, since energy debt does not measure effort, but the drainage of a resource. In that sense, energy debt interest is the amount of excessive energy consumed over time, that could be avoided if the issues were properly addressed earlier. In simple terms, it is the accumulated energy debt after n releases. This concept complements energy debt in the sense that it can be used to estimate the "real-world" cost of not fixing the smells, which can be monetary (as energy costs money) or uptime related (if the analyzed software is targeted for IoT/mobile devices).

Figure 5.4 illustrates our perception of energy interest, using our example from Figure 5.3. Again, 3 software releases are considered: $v1$, $v2$, and $v3$[3]. For this example, we are assuming that, as new versions are released, the issues from previous versions were not addressed. Hence, energy debt is always increasing. For $v1$, we consider that no interest was accumulated, due to the fact that energy debt depends on usage time. Hence, at the exact instant when the version was released, it was never used.

For release $v2$, we know that it added a new smell, $s3$. If **all** smells from the previous version ($v1$) were fixed upon release, then this $v2$'s minimum energy debt would be 5. However, since $v1$ contained smells, from the time interval comprised between the two releases, the software was excessively consuming 32 mJ for each minute it was being used (i.e., the energy debt from $v1$). Therefore, for release $v2$, the accumulated debt (i.e., the interest) is 32 mJ per minute.

When considering release $v3$, however, the reasoning to infer the interest requires adjust-

[3]The presented values refer to the minimum estimated debt

ments. For once, $v3$ has two predecessors, while $v2$ has only one. Between $v1$ and $v2$, the energy debt was 32, and between $v2$ and $v3$ it was 37. To estimate how much debt was accumulated, we should infer a value based on the two. One possible way to tackle this is to compute the average of **all** energy debts from previous releases. In this particular case, the minimum accumulated interest would be $\frac{(32+37)}{2} = 34.5$.

Finally, it is important to interpret interest similarly to how energy debt is interpreted: a minimum/maximum energy being excessively consumed **per usage time**. Hence, we argue that, when considering the interest for the n^{th} release, the expected usage time should be higher than the one for any previous release. Therefore it is guarantee that, even though energy debt is reduced from one release to another, the interest will be inflated for later releases.

5.3 Preliminary Experiment

At this point, we have already presented the concepts which define energy debt. In order to validate this concept, this section will describe an experiment performed over an existing software system, retrieved from GitHub, where we estimate the energy debt for all its releases. We start by presenting the software used for the experiment and the used approach for detecting the energy code smells (Section 5.3.1); afterwards, having detected the energy smells, we applied an energy debt analysis on each of the software's releases and present the obtained results (Section 5.3.2).

5.3.1 The Experimental Software System

In order to fully showcase the energy debt analysis described in the previous sections, we obtained a software system to use for a preliminary experiment, which followed two main requirements. First, it needed to be a system with several explicit releases. This way we could analyze the evolution of energy debt over time, and also present the energy interest values for each release. Second, it must be an Android application. This is due to the fact that 5 of the smells in our catalog are Android specific, and we wanted to explore the catalog as much as possible.

We searched for applications on GitHub, as it is a well-known and widely used platform to host open-source projects. Using GitHub's search engine, we searched for repositories with two characteristics: (i) the repository's main language should be `Java`, as it is the development

language for Android, and (ii) there should be specific references to the Android framework (i.e., `Java` imports to Android APIs, since we were filtering projects by code). GitHub's search engine only allows the retrieval of the first 1000 search results, hence we only selected the repositories referenced in those 1000 results containing several releases of Android applications.

With the filtered Android applications, the next step was to decide on which one to use for the experiment. Applications without smells had no interest to be analyzed, so defining an approach for detecting smells was necessary at this point. For this purpose, we used our previously developed tool, LintED[4], first referred to in Section 4.4 and thoroughly detailed in Section 6.4.

	versions									
	v0.4	#	**v0.5**	#	**v0.6**	#	**v0.7**	#	**v0.8**	#
smell count (#)	OLP	5	OLP	9 ▲	OLP	12 ▲	OLP	17 ▲	OLP	17
	EMC	1	EMC	2 ▲	EMC	1 ▼	EMC	1	EMC	1
					HMU	8	HMU	8	HMU	8
					MIM	2	MIM	3 ▲	MIM	3

Table 5.2: Detected smells on EscapeApp releases

After performing the lint analysis on all releases from the obtained Android applications, we selected the application with (i) a fluctuating number of smells per release and (ii) the highest number of smells detected in the latest release. The application that met this criteria was the EscapeApp[5], a client application used for a virtual reality escape game, which had 5 explicit releases. Table 5.2 presents the results of the lint analysis for this application. For each release, we indicate the number of occurrences of each detected smell, and how that number varied from the previous release.

5.3.2 Energy Debt Analysis

The smell detection performed onEscapeApp only provided the list of existing smells on each release. The lint tool does not offer a built-in mechanism to report the context of each detected smell, so information such as loop bounds could not be obtained automatically. As previously explained in Section 5.2.2, this information enhances the energy debt analysis. Thus we performed this analysis manually for each release and looked at each smell report to determine if it was being used witin a loop. If so, a loop bound value was inferred considering the loop's code.

[4]LintED webpage: https://gitlab.com/MarcoCouto/linted.
[5]EscapeApp code: https://github.com/alanvanrossum/kroketapp (visited on October 19, 2020).

Although we argue that it is not critical to have this analysis in order to estimate energy debt, for the SonarQube extension that we are working on, we intend to include the smell detection with context information regarding reachable paths and loop bounds inference to further enhance the analysis.

Based on the report presented in Table 5.2 and the manual code inspection performed, we can estimate the energy debt interval for each release. For instance, after inspecting each smell occurrence in release $v0.4$, we realized that the **EMC** smell was the only one found within a loop. After carefully inspecting and understanding the code, we estimated a loop bound of 10 iterations. Hence, the $cost_{min}$ and $cost_{max}$ values for release $v0.4$ can be calculated as follows:

$$cost_{min}(v0.4) = (5 \times E_{min}(OLP)) + (w_{EMC} \times E_{min}(EMC))$$
$$\Leftrightarrow cost_{min}(v0.4) = 5 \times 94 + 10 \times 557$$
$$\Leftrightarrow cost_{min}(v0.4) = 5{,}027\ mJ\ (per\ min)$$

$$cost_{max}(v0.4) = (5 \times E_{max}(OLP)) + E_{max}(EMC)$$
$$\Leftrightarrow cost_{max}(v0.4) = 5 \times 561 + 10 \times 9529$$
$$\Leftrightarrow cost_{max}(v0.4) = 98{,}095\ mJ\ (per\ min)$$

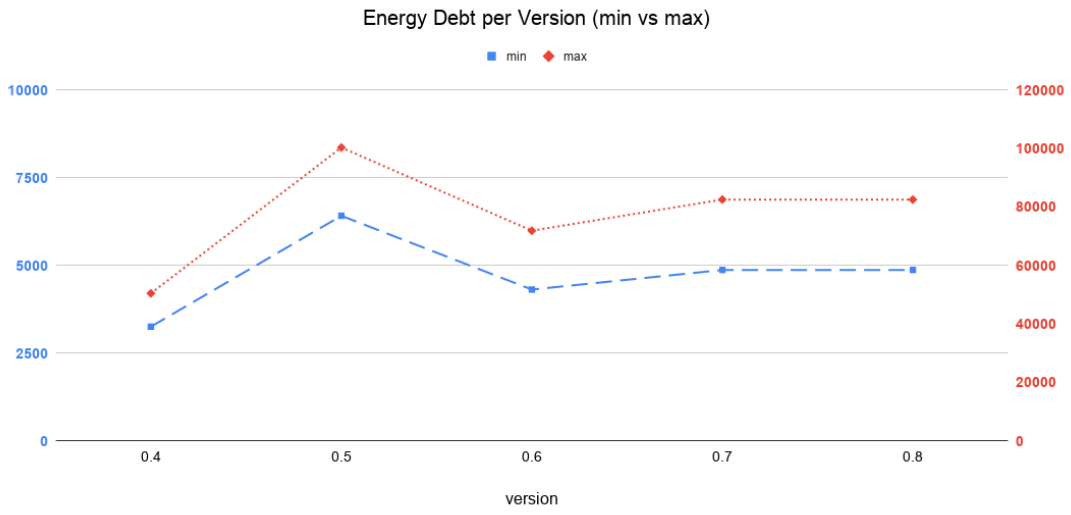


Figure 5.5: Minimum and Maximum Energy Debt for EscapeApp releases

The computed energy debt for each of the 5 versions is illustrated in Figure 5.5. As expected, the energy debt increases from release $v0.4$ to $v0.5$, as the total number of smells also increases. An interesting aspect, however, is the fact that the energy debt decreases after release $v0.5$. If we look at the detection report on Table 5.2, we see that it has 2 **EMC** smells. This smell has the highest value for maximum energy debt. It is also the smell with the highest weight, since it is always found inside loops. In that sense, addressing this smell before releasing $v0.6$ eventually payed off, even though other smells were introduced.

Nevertheless, it is important to also look at the concrete values of both the maximum and minimum energy debt. From release $v0.5$ to $v0.6$, energy debt might be reduced **at most** by $\approx 68\ J$, which is a considerable improvement. It is, however, the most profitable scenario in potential, and if we consider a more cautious analysis to the data (i.e., looking at the minimum energy debt variations) we can see that energy debt might only reduced by $\approx 4\ J$. Once again, it is up to the developers and/or product managers to decide how they look at the trade-off between refactoring effort and potential energy savings.

Estimating the accumulated interest for each release is also a straightforward task. If we consider, for example, releases $v0.5$ and 0.7, the minimum interest would be $6.04\ J$ per minute for the former, and $8.67\ J$ per minute for the latter, as the following calculations demonstrate:

$$interest(v0.5) = average(ed(v0.4)) = \left(6.04\ J\ ;\ 98.095\ J\right)$$

$$interest(v0.7) = average(ed(v0.4), ed(v0.5), ed(v0.6))$$

$$\Leftrightarrow interest(v0.7) = \left(8.67\ J\ ;\ 140.198\ J\right)$$

With these reference values, one could easily provide an estimate on how much energy was excessively consumed since the first release until release $v0.5/v.07$. Let us assume that the total amount of time an application is estimated to be used between these releases was, for instance, 5 full days (120 hours or $7,200$ minutes). This would mean that when $v0.5$ was released, the energy debt interest would vary between a minimum of $43,488\ J$, to a maximum of $706,284\ J$. When considering $v0.7$, with 3 prior releases, the expected usage time increases by: $3 \times 7,200 = 21,600$ minutes. Therefore, the interest values increase significantly as expected: a minimum of $187,272\ J$ and a maximum of $3,028,276\ J$. Even though this is merely a motivational case

study, it is safe to consider that neglecting the energy debt for several releases proves to be highly costly.

5.4 Conclusions

This chapter presented the concept of energy debt as the additional energy cost over time of a software system due to the occurrences of energy code smells in its source code. It is expressed as a function considering (i) the number of smells, (ii) the context in which they were detected, and (iii) the expected usage time of the application. Energy debt interest is also expressed as the accumulation of energy debt per release, which could be avoided by eliminating energy smells in previous releases.

Currently, we are extending an existing catalog of reported state-of-the-art energy code smells, and their known energy costs per usage time, which can be considered when calculating energy debt. Additionally, an extension of the concept of energy debt is being developed within the SonarQube framework, where it will support the inference of the context and number of detected smells, based on our catalog.

6 Multi Purpose Tools for Inspecting Energy of Programs

In this chapter, we provide a more in-depth explanation of each tool developed to support the contributions included in this thesis.

Each tool contains a description of the technologies used and how they were combined to fill the tool's purpose, as well as a discussion of every aspect of each tool's operations.

Although these tools were developed in a context-specific environment, they are intended to be modular, generic and multi-purpose. They are all still categorized as under development prototypes, yet we strongly believe they can be further enhanced and extended, and used by developers and other researchers.

6.1 Introduction

Energy analysis in software and energy-aware software development have become important concerns for software developers in recent years [Pinto et al., 2014, Pang et al., 2016]. This has become even more evident due to the growing evidence that end-users are continuously demanding for software to have more "battery-saving features" [Ath, 2017, Fri, 2019].

Addressing the challenge of energy-aware development is not, however, an easy task. Practitioners must either be well informed regarding existing recommendations for how software should operate (i.e., how it should be structured, and/or how its features should be implemented), or rely on existing tools that can help them analyze the energy footprint of their code. Nevertheless, the amount of knwoledge reported by researchers, in the form of energy-aware recommendations, is quite extensive and too verbose for developers to efficiently interpret (as we already discussed in Chapter 2). In addition, developers are also seeking for tools tht can not only profile the energy consumption of their software, but that can further help them improve their code, by automatically identifying potential energy hotspots and recommend less energy-greedy alternatives [Pang et al., 2016, Pinto and Castor, 2017].

With this chapter, we present the complete set of our prototype tools, developed in the context of the work from the previously presented chapters. These tools are mainly targeted for developers to use in order to examine the energy aspects of their code, and to help them understand how the code can be improved. Nevertheless, each and every of these tools was initially developed with the purpose of covering a specific usage context (e.g., a programming language, a target system/OS, a set of code patterns, etc.), which means that we invested some effort to make them modular and extensive. This was done with the purpose of helping researchers interested in our work, which may find helpfull to build on our tools and our findings and extend them, instead of starting from scratch.

6.2 C-LEM: Energy Model Creator for the CIL language

C-LEM is an energy model builder, currently capable of providing an energy estimation value for each instruction supported in the CIL language. Figure 6.1 shows the overall process for energy

modeling of both single instructions and functions.

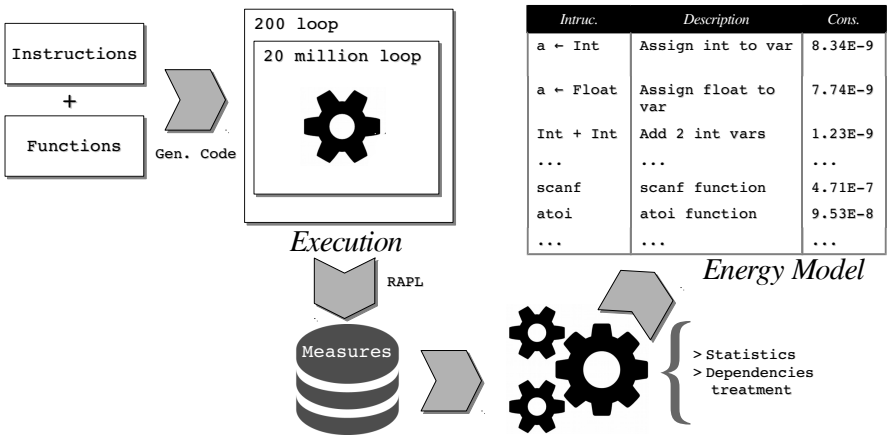


Figure 6.1: Workflow of C-LEM: the Energy Model Creator

All energy measurements reported by C-LEM are obtained using the RAPL tool, which as already discussed in Chapter 2 is know for providing highly accurate energy estimations [Hähnel et al., 2012]. At the moment, only CPU-related energy consumption is taken into consideration. Thus, it is the same as saying that all energy models created by this tool are actually CPU-based instruction-level energy models.

Modeling Process

The energy modeling process of single instructions varies between instruction types. Following the approach for energy modeling presented in [Hao et al., 2013], source code instructions are basically of two types: (1) single-cost operations or (2) API/function invocations. Single-cost operations are source code instructions, where its energy cost is constant, and it depends only on the type of their operands. Arithmetic and logic operations, such as a division or an ´´exclusive or", are examples of such instructions. The cost of such an operation consists of a function that maps the type of the operands involved to a constant value $\mathbb{C}$.

Estimating the energy cost of an API/function invocation requires a different approach. In most cases, we do not have access to the source code of an API/function, so we must have an estimation for its energy cost, which may depend on several things. First, it inevitably depends on the number of arguments of the function, as they need to be pushed to the call stack before the function call. Second, it can depend on the value of the arguments, since there might be a loop in the function parameterized by the arguments, or it may have a premature return instruction

included, which can be triggered depending on one or more arguments. The type of the arguments may also play a role in the energy cost (ex. reading and storing a double is different than reading and storing an integer.)

In order to address these problems, we had to look at each one individually. For functions with a variable number of arguments or types (such as `scanf`), we tested them several times, each time with an increasing number of arguments, and with different types. After obtaining the consumption values, the tool tries to detect if there is a relation between the number of arguments (of the same type) and the energy consumed, i.e., if diving the consumption for the number of arguments results in an equal value $\mathbb{C}$, or in values with a negligible difference between them. If so, the consumption value for that function will then be the constant value $\mathbb{C}$ multiplied by the number of arguments. If the consumption of a function is proportional not to the number of arguments, but to the length of one or more arguments (e.g., the `strlen` function will behave differently for strings with different sizes), than we take the highest consumption observed, regardless of the argument. We do so since it is not possible to estimate the length of data structures, such as arrays of matrices, through static analysis, and so we have chosen a conservative approach that fits our needs.

The workflow for estimating the energy cost of both single-cost and API/function calls is identical. For a given instruction `i`, the tool executes it 20 million times. This ensures a measurement of a sufficiently long duration that exceeds the sampling interval of our energy measurement tool (RAPL). We repeated this process 200 times, to reduce the impact of cold starts, cache effects or background processes, and we obtain the energy consumption for each time. Using these values, we calculate the average value, after removing the outliers (5 highest and lowest values).

In most cases, single-cost instructions (or even functions) depend on other instructions. For example, the instruction `a=b*2` is an assignment to the variable `a` of a multiplication between the variable `b` and the constant value `2`. In order for this to work, the variable `b` needs to be already declared and have some value assigned to it, but we only want to know the cost of `a=b*2`. To address this problem, we have assigned, to each instruction/function, the list of independent instructions they depend on. Therefore, after measuring the energy consumed by each instruction/function, we subtract to it the previously obtained consumption for each dependency. Hence, the final value will reflect the energy consumption for the instruction/function only. Once again, this is in full compliance with the modeling approach presented in [Hao et al., 2013].

Given the average value for each instruction/function `i`, calculated after the 200 measurements, we subtract the average value of each of the dependencies. The final consumption will be the resulting value.

Tool Inputs and Workflow

C-LEM requires two components to be defined beforehand. The first one is the **test controller** file, which is where the code for **all** instructions to be tested are included. Each one must be implemented in its own function. The main function of this file must act as a controller, compiling only the function of the instruction to be tested. This is done by including conditional compilation directives for each function, where the the macro identifier is unique for each one. The following snippet shows one such example, where the instruction to be tested is the assignment of an constant integer value to a variable.

```
1  ...
2
3  #ifdef ASS_VAR_INT
4    val = assign_var_int();
5  #endif
6
7  ...
```

The second required component is the list of instructions/functions to analyze. Each element on this list is a JSON object, where 3 attributes must be defined:

- the `'code'`: a unique value, to identify the instruction/function;
- the `'cpp'` code: the macro identifier to be used by the **test controller** to compile the correct instruction's source code;
- the `'dependencies'` list: since some instructions depend on others, the `'codes'` included in this list will tell C-LEM what consumption values from other instructions must be subtracted after testing and measuring.

The following example shows the definition of two instructions, where the second depends on the first one (all instructions have a dependency code 0, which translates to the energy cost of the **test controller** calling the instruction's function).

For API/function calls, three additional attributes are required: the name of the functions, the list of header files required to be included, and the types of the arguments. C-LEM will use this information to include necessary dependencies and define different function call instructions.

```
...
{
  "code" : 4,
  "cpp" : "ASS_VAR_INT",
  "dependencies" : [0]
},
...
{
  "code" : 16,
  "cpp" : "ASS_ADD_VAR_INT",
  "dependencies" : [0,4]
}
...
```

The arguments will always be automatically and randomly generated, according to the types definitions.

Once these two components are defined, C-LEM will start executing and measuring the energy consumed by all instructions in the order provided, starting with single-cost operations and then moving to functions. After all instructions are tested, it will subtract the consumption of dependencies, as explained before. Finally, the obtained values are saved and stored in a file.

6.3 Serapis: Estimate WCEC through Program Analysis and Energy Models

Serapis is a prototype implementation of our WCEC technique. All the core functionalities are implemented in Haskell, and it is directly dependent on the energy model provided by C-LEM. Next, we explain the implementation decisions and challenges we faced, and we explain the its workflow.

Preparing the Code

In this step, Serapis prepares a bundle file with all the code relative to a SPL, making it compatible with the CFG transformation function. We should point out that, as previously discussed, our approach to build the CFG representing a SPL and its products is based on parsing the code to an AST representation and then transform it to a CFG. Considering an AST representation based on traditional C code would result in a highly complex CFG transformation procedure, which is not desirable for a prototype. Hence, in order to simplify this process, we considered an intermediate

representation of the C language, called **C Intermediate Language - CIL** [1]. This language works as a subset of the C language, making it possible to have all the functionalities allowed by the C language but reducing the syntactic sugar to the minimum (for example, for loops and variable initialization at declaration time are not allowed).

To start the code bundling and treatment procedure, Serapis requires the SPL code directory and the path to the main file. With these two arguments provided, it then bundles all the code inside the main file. By examining the `include` statements on each file, starting from the main one and using a recursive strategy, we are sure to include only the required local dependencies. This file is then passed to the feature-based preprocessor, which will handle the conditional compilation directives that restrict code to certain features.

The preprocessor references to features (as explained in Section 3.2.1) also need to be included in order to identify code blocks belonging to specific features. Since such references are not parsable by CIL, we developed a domain specific preprocessor capable of transforming such references into parsable statements. Basically, it transforms every `#ifdef <feature>` primitive in a function call, in the form `__feature_<feature>()`, and every `#endif` in a function call in, the the form `__endFeature()`. At the CFG construction phase, every function call is initially checked to determine if it restricts the following code to certain features or if it is code shared by all products.

Using an existing interface that supports CIL code analysis[2], Serapis is then able to load the CIL code bundle file, parse it and create the AST representing the file (the **CIL file AST**). The next task consists of looking for the AST subtree which represents the main function, and transform it into a **whole program AST**. This is done by recursively finding and replacing function calls' nodes with the AST representation of their respective source code, whenever it is possible (e.g., for system calls like `printf` it will not find the code, and will maintain the statement as it is). The name of the function parameters is replaced with the corresponding argument name passed to the function call, once again following a recursive strategy.

The resulting AST can then be transformed into a feature-sensitive CFG, using a top-down approach. If a function call appears in the form `__feature_feature-A()`, the AST nodes which follow will result in CFG nodes with `feature-A` included in the list of features where

[1] More information can be found here: https://people.eecs.berkeley.edu/ necula/cil/ (visited on October 19, 2020).

[2] The CIL package: https://hackage.haskell.org/package/cil (visited on October 19, 2020).

the node must be considered. Once the corresponding `__endFeature()` call appears in the AST, `feature-A` will stop being included in the following nodes. The CFG creation function was developed using an Haskell library for graph creation and manipulation[3].

The overview of the preprocessing and CFG construction steps is depicted in Figure 6.2.

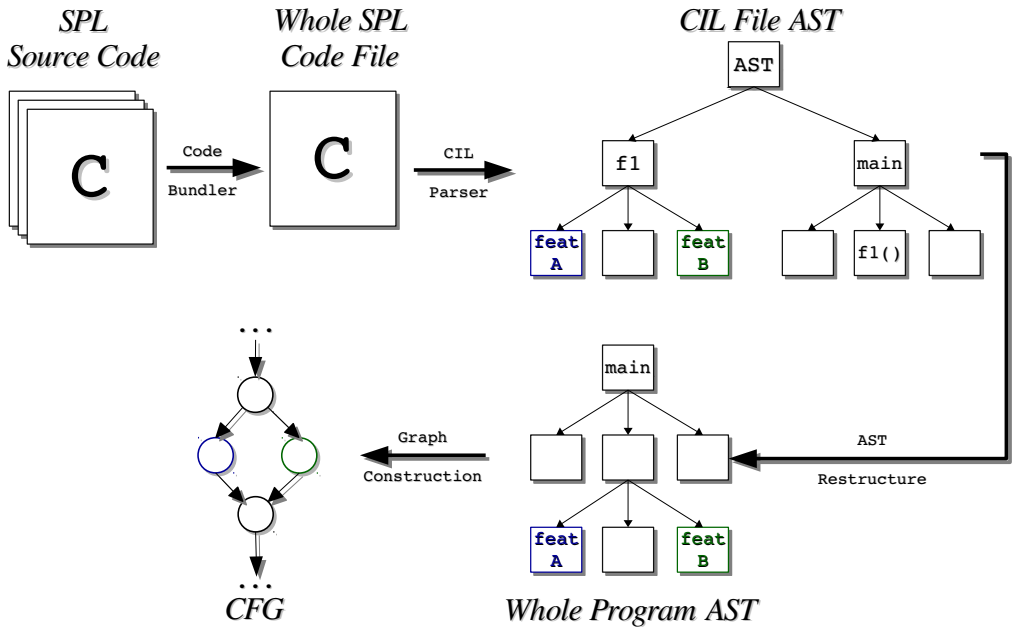


Figure 6.2: Overview of the Preprocessing and CFG Creation Steps

Performing WCEC Estimation

The generated CFG from the previous step is the input to the WCEC estimation step. Here, Serapis performs all required static analysis with the CFG (machine behavior, dataflow, loop bounding, etc.). Each node will then contain the result of each static analysis until that point. Finally, a function is applied to each node on the CFG, called the **local energy bounding** function. It uses the computed state on each node, and determines the maximum energy consumption that the node would consume. Naturally, this value also depends on the predecessor nodes as well, since for some dataflow analysis the result of a node is propagated to their successors (commonly referred to as forward analysis [Møller and Schwartzbach, 2015]).

Once all the local energy bounds are computed, the resulting CFG is passed to the WCEC prediction function. Here, the CFG and the loop upper bounds are used to create the constraints and the maximization function needed by IPET (as described in Section 3.3.4). For the maximiza-

[3]The functional graph libray (fgl): https://hackage.haskell.org/package/fgl (visited on October 19, 2020).

tion function, the cost of each statement is given by the energy model, along with the information from the other analysis. To implement this, we used an integer linear programming (ILP) library for Haskell, which supports the creation of a maximization function along with a set of constraints, and has an implemented algorithm to compute the solution for such a problem [4].

Serapis reports, for each product within the analyzed SPL, the WCEC estimation, the features that characterize the product, and the expected instructions to execute in the WCEC execution path. The following code snippet shows an example of this report for 2 products which are part of the MinePump SPL (introduced in Chapter 3):

```
[{
  "WCEC": 0.16637651085850022,
  "features": [
    "HIGH_WATER_SENSOR",
    "LOW_WATER_SENSOR"
  ],
  "instructions": {
    "ASS_VAR_INT": 1850027,
    "ASS_ADD_VAR_INT": 300003,
    "FUN_printf": 500010,
    "ASS_SUB_VAR_INT": 500008,
    "ASS_VAR_VAR_INT": 7000078,
    "CMP_VAR_INT": 1500017
  }
},
{
  "WCEC": 0.1561803833997386,
  "features": [
    "HIGH_WATER_SENSOR"
  ],
  "instructions": {
    "ASS_VAR_INT": 1150019,
    "ASS_ADD_VAR_INT": 300003,
    "FUN_printf": 500010,
    "ASS_SUB_VAR_INT": 500008,
    "ASS_VAR_VAR_INT": 4200046,
    "CMP_VAR_INT": 1150013
  }
},
...
```

Our tool shares all the advantages of the monotone framework for static analysis [Møller and Schwartzbach, 2015], thus the correctness properties are maintained. Its accuracy can only be increased by improving both the energy and machine behavior models. Nevertheless, it also has the same limitations. The real unavoidable limitation of static analysis is the fact that recursive functions cannot be considered. Therefore, only SPLs without any recursive functions can be evaluated by it.

[4] The ILP package: https://hackage.haskell.org/package/hmatrix-glpk-0.5.0.0 (visited on October 19, 2020).

Moreover, the dataflow analysis process, used to determine the loop upper bounds, is very costly in terms of execution time and memory usage. For an example with around 500 loops and 1500 variables, our tool gave accurate upper bounds, but it did not end in an acceptable time (took several almost 2 days to finish), so it was not convenient to use this approach while testing and refining the tool. Given that, we inspected all the loops in the analyzed SPLs to see what was the worst case scenario in terms of number of executions, and we manually assigned that value to each loop. However, we have designed some improvements to implement in the dataflow analysis step, in order to significantly reduce the time it takes to obtain the upper bounds. Such improvements include memoization and exclusion of variables not used in the loop conditions.

At the moment, Serapis[5] has around 2330 lines of code, divided in 11 modules.

6.4 LintED: A Lint-based EGAP Detector

LintED consists of a collection of files that express custom lint-based detection rules. Currently, it supports the detection of any of the 11 EGAPs firstly introduced in Section 4.2. Its main advantage when compared to similar pattern detection tools, such as (for instance) aDoctor [Palomba et al., 2017], is that it is essentially an extension of Android lint, which is an Android SDK tool. This means that its evolution and maintenance is easier, as it only requires existing detection rules to be adapted/improved, or new rules to be introduced, since the whole detection mechanism is maintained/evolved by the Android SDK developers team.

Next, we explain the essence of the detection rules (i.e., how they are defined and what exactly do they report), and how the tool itself integrates with Android lint in order to examine an application.

Defining a Detection Rule

Android lint already ships with several detection rules, grouped in different context categories, such as performance or Security. These are known as the default rules. Defining a custom lint rule can be made by extending an existing default one, or creating one from scratch. The latter approach consists of creating a Java class which extends the abstract scanning class and defines what is called the rule issue object, which consists of the following properties:

[5] Serapis is available at https://gitlab.com/MarcoCouto/serapis (visited on October 19, 2020).

- **Id**: A unique identifier, typically the name of the pattern to be detected;
- **Description**: A short description of what the rule is examining;
- **Explanation**: A rationale expressing why the pattern being detected is or can be an issue;
- **Category**: The id of an existing lint category;
- **Priority**: The priority level value (from 1 to 10);
- **Severity**: How severe is the pattern detected (warning, error, fatal, etc.);
- **Implementation**: The actual implementation class, which typically is the issue defining class itself.

The detector class must also define the rule's workflow: what AST node types must be analyzed, what to do before/after traversing through the AST, and most importantly what visitor to use in each traversal and for each node. A visitor is a method named visit, which only input is an object with the type of the AST node to analyze. The detector traverses through the AST in a bottom-up approach, running all visit methods whenever it finds a node signaled as analyzable. All the EGAP-related properties are inferred in visit methods; when an EGAP is found we use the built-in report system to store its location. The next snippet shows how we implemented one such method, namely to detect the `HashMapUsage` EGAP, and that reports when a method variable is declared with the HashMap type:

```
1  private void checkAndReport(PsiVariable var) {
2    if (var.getType() == null) return;
3    String varType = var.getType().getCanonicalText();
4    PsiExpression init = var.getInitializer();
5
6      if (varType.startsWith(mHashMapClass)) {
7        Reporter.reportIssue(mContext, ISSUE, var);
8      } else if (varType.startsWith(mMapClass)
9        && init != null && init.getType() != null) {
10         String initTp = init.getType().getCanonicalText();
11         if (initTp.startsWith(mHashMapClass)) {
12           Reporter.reportIssue(mContext, ISSUE, var);
13         }
14 }  }
```

As we can see, this method will perform a series of checks on the all AST nodes representing a method variable. If one of them contains a reference to HashMap in the variable's type, it will emit a message using the Reporter class, which will tell Android lint that an issue was found wherever the "faulty" variable is declared (i.e., the file and source line). Android lint will be responsible to generate the result detection file, indicating the type and location of every detected EGAP.

Linking LintED with Android lint

Running the `build` task on LintED's source code will do all that is necessary for lint to be able to run the custom detection rules. First, it will generate a JAR file, containing the compiled code. Second, it places this file in a specific location in the file system, where lint expects the rules to be and where it has permission to access and use the files. With this done, it is then necessary to tell lint which rules to run in a given project.

By omission, lint will run all the default detection rules. In order to redefine this procedure, it is necessary to change an application's build file (i.e., the `build.gradle` file) and change the lint options, as described in the following snippet:

```
1 android {
2   ...
3
4   lintOptions {
5     check "DrawAllocation", "Wakelock", "Recycle", ...
6     abortOnError false
7   }
8   ...
9 }
```

The values specified on line 6 are the issues' ids, as described earlier. The prefix `'check'` indicates that lint will only run the detection rules of whatever issues are defined after.

After finishing scanning for EGAPs, lint generates an output XML file with everything reported by the detection rules. Each entry in this files contains the EGAP id, the description and explanation, as well as the file path and the line number where the EGAP was detected. The following snippet shows a partial output, produced for an application with 2 EGAPs found, `Recycle` and `ExcessiveMethodCalls`:

```
1  <issues format="4" by="lint 26.1.1">
2      <issue
3          id="Recycle"
4          summary="Missing `recycle()` calls"
5          ...
6      </issue>
7      <issue
8          id="ExcessiveMethodCalls"
9          summary="Number of method calls can be reduced"
10         ...
11     </issue>
12 </issues>
```

6.5 Chimera: An EGAP-oriented Testing Tool for Android

Chimera is an EGAP-oriented automatic energy testing framework. Essentially, this framework is suited for testing the energy impact of replacing an EGAP with an alternative, in order to assess if the alternative actually reduces the amount of energy consumed. It is written in Python, and developed to provide its features with a command line interface (CLI). As explained in Section 4.5, Chimera can be described as a main CLI file (the pipeline script), which is responsible for controlling 4 components. Nevertheless, the component's code is modular, and for some of them there is a CLI file created to simplify its usage.

Chimera's Architecture and Features

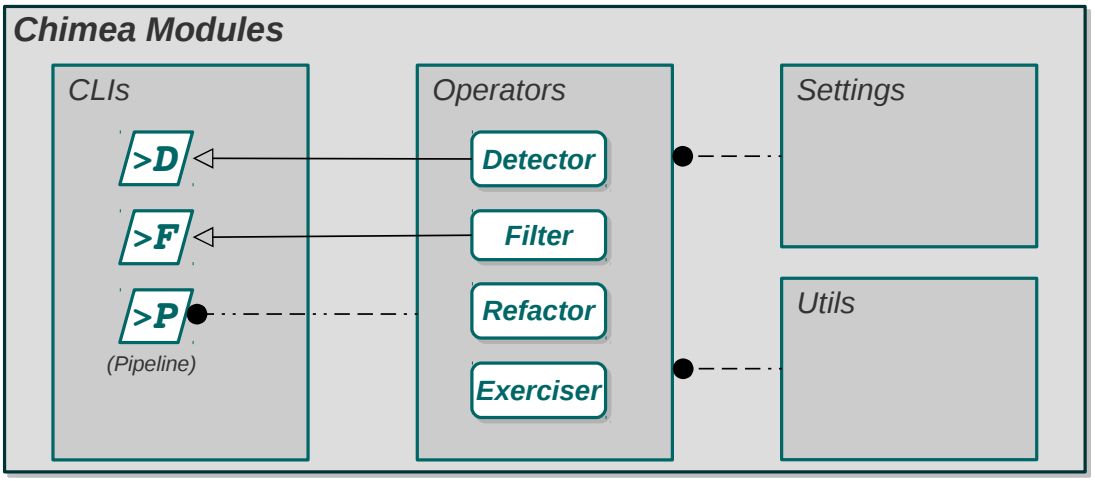


Figure 6.3: Chimera's Architecture

Figure 6.3 shows the top level overview of Chimera's modules architecture. Essentially, Chimera is composed of 4 modules:

- **Operators**: code relative to each operator is stored.
- **Settings**: files containing properties definitions (e.g., paths, flags, names of EGAPs, etc.).
- **Utils**: code relative to functions reused in different components (e.g., load/store functions, file parsing, etc.).
- **CLIs**: the CLI scripts.

```
$ ./filter_cli.py                \\
        "(l:HashMapUsage,>=,1)" \\
        "(l:Recycle,>=,3)"      \\
        -o "app id"
```

The operators are the core entities of Chimera's features. The Detector operator is responsible for inspecting one or more applications, and report for each one how many EGAPs (and which ones) were found. This task is performed using the previously described LintED tool. This operator expects as input the path to the folder where the applications to analyze are stored. This path can be in a remote machine, provided that this is specified in the `'remote.py'` file, inside the **Settings** module. It returns as output a Python dictionary, where the id of an application is mapped to the list of EGAPs found in it. The CLI of this operator basically runs this procedure on a repository passed as argument, while showing the progress and logging in what applications the Detector had issues.

Filter is the operator responsible to use the reports from the Detector and the provide the list of applications where a given set of EGAPs was found. Hence, it expects as input a list of EGAPs (i.e., a combination), and the object returned by the Detector. The output is a filtered Python dictionary. The CLI of this operator provides an interface to query the Detector results, and obtain the results in a pretty-printed and easily interpretable output. This filter mechanism in this CLI allows queries to define boundaries for each filtered EGAP. For instance, the following command will look for applications with 1 or more `HashMapUsage` and 3 or more `Recycle` EGAPs found, and it will only print the id of applications that match this query (omitting all other details):

The purpose of the Refactor operator is to apply a set of the refactoring rules, implemented using AutoRefactor, to the code of an application. The path to the code and the list of refactorings to apply are required as input. Refactor will attempt to apply all refactorings, and rely on the output provided by AutoRefactor to verify if there was no issue in this task (e.g., an EGAP marked to be refactored could not be found in the code). If an issue occurs, this operator will return an empty list and a flag with a value different from zero. If everything goes as expected, the return flag will be zero, and the list will express how many refactorings where performed (and which ones).

Finally, the Exerciser operator is the one that orders the execution of a set of usage scenarios on a given application, while collecting data regarding energy consumption and execution time

on each usage scenario. Hence, the workflow of this component consists of (i) testing if there is an Android device connected to the machine running Chimera, (ii) installing the application (and removing old versions installed, if any), (iii) start/stop the energy and execution time profilers before/after running a usage scenario, and (iv) examine the resulting files, returning the energy and execution time values for each usage scenario.

Usage Scenarios

To define an usage scenarios, we used the Android MonkeyRunner[6]. We chose this tool because it provides a very convenient method to inject events on an application, and by this to simulate usage scenarios without knowing the application context. In fact, a recent study shows that randomly injecting events over Android applications provides a way of testing them and assure the best ratio between coverage and effort needed for the setup [Choudhary et al., 2015].

Considering we are analyzing a substantial set of applications, it would be unpractical to build context-dependent usage scenarios for each. Therefore, we designed **25 usage scenarios** that can be applied to any application. Each scenario consists of 2 parameters that are given to MonkeyRunner: the application package, and a randomly generated sequence of **25** events to be triggered. Each sequence is unique, in order to simulate different execution paths of the application. The scenarios run on the original and the refactored version are exactly the same.

In each scenario execution, Exerciser opens the application, waits **5 seconds** (warm-up), and then starts the MonkeyRunner events and the measurement procedures. When MonkeyRunner finishes, Exerciser stops measuring, closes the application, stores the collected values, and waits another **5 seconds** (cool-down). This procedure is repeated both for the original and the refactored versions of an application, and it intended to reduce the cold/warm cache effects on the results.

Reusing and Evolving Chimera

Every setting in the framework is completely configurable. This means that, for instance, one can choose how many usage scenarios to run, how many times a scenario should be repeated, or the warm-up and cool-down times. It is even possible to replace external tools, such as AutoRefactor and LintED, for alternatives suited for the same purposes. They however need to provide an

[6]Android MonkeyRunner webpage: http://bit.ly/2VzPFAF (visited on October 19, 2020).

interface compatible with the respective operator.

In order to Chimera to support new EGAPs, the external components responsible for detecting and refactoring EGAPs need to be improved as well. In other words, there must be a new refactoring rule and a new detection rule for each new EGAP. An alternative would be to use tools that already provide these mechanisms. Once this is addressed, all there is to do is to include the EGAP name in the `'egaps'` file, included in the Settings module.

7

Conclusions

In this chapter, we summarize the research works presented and discussed in the previous sections, and share our conclusions on the work done. We look back at the research goals introduced in the beginning of this dissertation, providing an overview of how we addressed them.

Finally, we present a few research paths that can be explored in the future, both by ourselves or other researchers working in the area.

In this thesis, we presented different techniques and tools that can aid developers (and other practitioners) in examining the energy efficiency of their software systems. We also explored and expanded state-of-the-art approaches, which main goal was to help reduce the energy consumed by Android applications, in order to understand the general applicability of their results. During the results discussion, we further elaborated on the issues associated with carelessly addressing every potential energy inefficient code, as well as the negative impact of leaving them totally unattended for too long.

As discussed before, the main goal of the core contributions incorporated in this thesis was to address a lead research question, introduced in Section 1.1:

Main Research Question

How can we increase the knowledge that is available to practitioners, so that they can accurately assess and improve the energy efficiency of their programs/applications?

Our main strategy do address this research question was to develop new techniques, and extend the existing knowledge base, so that practitioners could examine their software without the need to dynamically test it. In other words, we aimed at providing mechanisms to enable a **static reasoning about energy efficiency** of software. In order to do so, we divided the main research question into three independent goals, which were presented in Section 1.1. Next, we summarize, for each research goal, the work we developed, while outlining the main findings we obtained.

Research Goal 1

Accurately predict the energy consumed by several similar software systems.

The WCEC approach we presented in Chapter 3 is capable of providing accurate upper-bounded estimations for the energy consumed by a program, in the worst case scenario. Moreover, our prototype tool, Serapis, is also capable of performing a WCEC analysis for several products within a SPL. The results obtained from the evaluation study show that:

1. our technique always provides a slightly overestimated energy consumption prediction, which was a required property from the beginning;
2. the analysis time of our tool is shorter than the amount of time required to fully test each

product within a SPL individually, hence showing the advantages of using a static approach when inspecting multiple programs/systems;

Given the simplicity of the employed models (i.e., energy and machine behavior), and the fact that our tool is yet a prototype with potential for improvements, we strongly believe that this work can be seen as a foundation for other works focused on providing accurate energy predictions.

Research Goal 2

Study and understand how documented energy improvements behave when applied together, and in a wide variety of contexts.

With the work presented in Chapter 4, we further explored all the state-of-the-art energy-greedy Android patterns. We performed a large scale study, where all the patterns were tested under the same conditions, with multiple execution scenarios. Additionally, we also examined how is energy consumption affected when applying multiple patterns at the same time. This work allowed us to conclude that:

1. replacing an energy-greedy pattern generally results in consistent energy savings;
2. the energy savings are typically very different among patterns, and for some patterns the savings highly depend on the execution context;
3. for a few combinations of patterns, replacing all of them actually has a more negative than positive impact on energy savings.

Research Goal 3

Develop an approach which can relate energy analysis/improvement with software evolution and maintenance.

The novel Energy Debt concept we introduced in Chapter 5 represents a different way of analyzing energy in software. It relates energy greedy code detection with software evolution, in the sense that it allows practitioners to understand the impact of fixing such code smells in different evolutionary steps of a software system (i.e., releases).

We performed a preliminary/motivational experiment, where we examined the existence of energy-greedy patterns (presented in Chapter 4) in an Android application. The observations from the experiment reflect the main idea behind this concept: it is possible to forecast the energy behavior of an application, by looking at how the energy greedy code evolves. We believe this might help practitioners prioritize their maintenance effort.

7.1 Future Work

With this thesis, we addressed two particular research topics: **energy estimation** and **energy-aware software evolution**. While the works presented here have contributed to further enhance the research area that it **Energy Analysis in Software**, there is still room for improving and extending our works.

The WCEC technique detailed in Chapter 3 can be applied to other contexts. Extending it to other languages, or to a lower abstraction level (e.g. assembly or LLVM), would make the technique reachable to other software systems. It would also be interesting to test this approach on software targeted for embedded systems, where the impact of determining accurate energy consumption upper bounds could be higher. The effort to expand the applications of WCEC would always involve developing new energy/behavior models, and building adequate mechanisms to parse a program's AST and transform it into a CFG. Finally, it would also be interesting to extend WCEC so it could provide not only a single estimation, but a distribution of estimations.

The EGAPs described in Chapter 4 are the current existing ones for Android. Nevertheless, we strongly believe that new energy-greedy patterns will eventually arise, hence it would be interesting to further extend our work with a similar analysis for such patterns. Moreover, considering existing EGAPs in other contexts (i.e., languages or paradigms), and perform a similar study to the one we did in Section 4.5, could result in valuable new findings. This would aid developers working with other technologies. Finally, the study itself could be further enhanced. We are still not fully aware of why refactorings do not produced energy savings in all cases. Collecting additional data, such as method call trace or memory usage, could provide additional findings, although the study would need to be fully repeated, which would take a considerable amount of time.

The Energy Debt concept presented in Chapter 5 is independent of any language, paradigm, or system. Nevertheless, it requires an energy smell catalog, and a smell detection tool. We demonstrated how to easily instantiate this concept for Android, using the findings from our Chapter 4, and we are currently developing a prototype tool that encompasses all these notions and mechanisms. Nevertheless, in Chapter 4 we proved that combining the refactorings of multiple EGAPs may result in unexpected results, as the savings can sometimes be reduced or even annulled. It would certainly be interesting to understand how the findings regarding combinations of refactorings can be combined with the energy debt concept, as to improve its reliability.

7 References

[Val,] Valgrind webpage. https://valgrind.org/. Accessed: 2020-09-20.

[Ath, 2017] (2017). Camera and battery features continue to drive consumer satisfaction of smartphones in US. https://www.strategyanalytics.com/strategy-analytics/news/strategy-analytics-press-releases/2017/05/31/camera-and-battery-features-continue-to-drive-consumer-satisfaction-of-smartphones-in-us. Accessed: 2018-01-24.

[Rda, 2017] (2017). Why energy is a big and rapidly growing problem for data centers. https://www.forbes.com/sites/forbestechcouncil/2017/12/15/why-energy-is-a-big-and-rapidly-growing-problem-for-data-centers/#43c301b75a30. Accessed: 2019-07-25.

[ICT, 2018] (2018). Ict could consume up to 20% of electricity in 2030. https://www.enerdata.net/publications/executive-briefing/expected-world-energy-consumption-increase-from-digitalization.html. Accessed: 2019-07-25.

[Jha, 2018] (2018). Our phones and gadgets are now endangering the planet. https://www.theguardian.com/commentisfree/2018/jul/17/internet-climate-carbon-footprint-data-centres. Accessed: 2018-01-24.

[Per, 2018] (2018). Perf linux manual page. https://man7.org/linux/man-pages/man1/perf.1.html. Accessed: 2020-09-20.

[Fri, 2019] (2019). The most wanted smartphone features. https://www.statista.com/chart/5995/the-most-wanted-smartphone-features. Accessed: 2018-01-24.

[Ene, 2019] (2019). World energy consumption statistics. https://yearbook.enerdata.net/total-energy/world-consumption-statistics.html. Accessed: 2019-07-25.

[Allman, 2012] Allman, E. (2012). Managing technical debt. Commun. ACM, 55(5):50–55.

[Ampatzoglou et al., 2015] Ampatzoglou, A., Ampatzoglou, A., Avgeriou, P., and Chatzigeorgiou, A. (2015). Establishing a framework for managing interest in technical debt. In Proceedings of the Fifth International Symposium on Business Modeling and Software Design - Volume 1: BMSD„ pages 75–85. INSTICC, SciTePress.

[Ampatzoglou et al., 2018] Ampatzoglou, A., Michailidis, A., Sarikyriakidis, C., Ampatzoglou, A., Chatzigeorgiou, A., and Avgeriou, P. (2018). A framework for managing interest in technical debt: An industrial validation. In Proceedings of the 2018 International Conference on Technical Debt, TechDebt '18, page 115–124. Association for Computing Machinery.

[Apel et al., 2013] Apel, S., Rhein, A. v., Wendler, P., Größlinger, A., and Beyer, D. (2013). Strategies for product-line verification: Case studies and experiments. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 482–491. IEEE Press.

[Apel et al., 2011] Apel, S., Speidel, H., Wendler, P., von Rhein, A., and Beyer, D. (2011). Detection of feature interactions using feature-aware verification. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 372–375. IEEE Computer Society.

[Banerjee et al., 2014] Banerjee, A., Chong, L. K., Chattopadhyay, S., and Roychoudhury, A. (2014). Detecting energy bugs and hotspots in mobile apps. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, page 588–598. Association for Computing Machinery.

[Bodden et al., 2013] Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., and Mezini, M. (2013). Spllift: Statically analyzing software product lines in minutes instead of years. In Proc. of 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '13, pages 355–364. ACM.

[Brabrand et al., 2012] Brabrand, C., Ribeiro, M., Tolêdo, T., and Borba, P. (2012). Intraprocedural dataflow analysis for software product lines. In Proc. of 11th Annual Int. Conf. on Aspect-oriented Software Development, AOSD '12, pages 13–24. ACM.

[Capilla et al., 2014] Capilla, R., Bosch, J., Trinidad, P., Ruiz-Cortés, A., and Hinchey, M. (2014). An overview of dynamic software product line architectures and techniques: Observations from research and industry. Journal of Systems and Software, 91:3 – 23.

[Carette et al., 2017] Carette, A., Younes, M. A. A., Hecht, G., Moha, N., and Rouvoy, R. (2017). Investigating the energy impact of Android smells. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 115–126.

[Chatzigeorgiou et al., 2015] Chatzigeorgiou, A., Ampatzoglou, A., Ampatzoglou, A., and Amanatidis, T. (2015). Estimating the breaking point for technical debt. In 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), pages 53–56.

[Choudhary et al., 2015] Choudhary, S. R., Gorla, A., and Orso, A. (2015). Automated Test Input Generation for Android: Are We There Yet? (E). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE '15, pages 429–440. IEEE Computer Society.

[Chowdhury et al., 2018] Chowdhury, S., Di Nardo, S., Hindle, A., and Jiang, Z. M. (2018). An exploratory study on assessing the energy impact of logging on android applications. Empirical Software Engineering, 23(3):1422–1456.

[Chowdhury and Hindle, 2016] Chowdhury, S. A. and Hindle, A. (2016). Greenoracle: Estimating software energy consumption with energy measurement corpora. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, page 49–60. Association for Computing Machinery.

[Chowdhury et al., 2019] Chowdhury, S. A., Hindle, A., Kazman, R., Shuto, T., Matsui, K., and Kamei, Y. (2019). Greenbundle: An empirical study on the energy impact of bundled processing. In Proceedings of the 41st International Conference on Software Engineering, ICSE '19, page 1107–1118. IEEE Press.

[Colin and Lucia, 2016] Colin, A. and Lucia, B. (2016). Chain: Tasks and channels for reliable intermittent programs. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, page 514–530. Association for Computing Machinery.

[Colin and Lucia, 2018] Colin, A. and Lucia, B. (2018). Termination checking and task decomposition for task-based intermittent programs. In Proceedings of the 27th International Conference on Compiler Construction, CC 2018, page 116–127. Association for Computing Machinery.

[Costa-Requena et al., 2014] Costa-Requena, J., Peuhkuri, M., and Manner, J. (2014). Analysis of software and server hardware impact on energy efficiency. In Energy Conf. (ENERGYCON), 2014 IEEE International, pages 716–721.

[Couto et al., 2017a] Couto, M., Borba, P., Cunha, J., Fernandes, J. P., Pereira, R., and Saraiva, J. (2017a). Products Go Green: Worst-Case Energy Consumption in Software Product Lines. In Proceedings of the 21st International Systems and Software Product Line Conference - Volume A, SPLC '17, pages 84–93. ACM.

[Couto et al., 2017b] Couto, M., Pereira, R., Ribeiro, F., Rua, R., and Saraiva, J. (2017b). Towards a Green Ranking for Programming Languages. In Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP 2017, pages 7:1–7:8. ACM.

[Couto et al., 2020a] Couto, M., Saraiva, J., and Fernandes, J. P. (2020a). Energy refactorings for android in the large and in the wild. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 217–228.

[Couto et al., 2020b] Couto, M., Saraiva, J., and Fernandes, J. P. (2020b). Statically analyzing the energy efficiency of spls. IEEE Transactions on Sustainable Computing. (submitted).

[Couto et al., 2014] Couto, M., T., C., Cunha, J., Fernandes, J. P., and Saraiva, J. (2014). Detecting Anomalous Energy Consumption in Android Applications. In Quintão Pereira, F. M., editor, Programming Languages, volume 8771 of LNCS, pages 77–91. Springer Int. Publishing.

[Couto et al., 2011] Couto, M. V., Valente, M. T., and Figueiredo, E. (2011). Extracting software product lines: A case study using conditional compilation. In 2011 15th European Conference on Software Maintenance and Reengineering, pages 191–200.

[Cruz and Abreu, 2017] Cruz, L. and Abreu, R. (2017). Performance-based Guidelines for Energy Efficient Mobile Applications. In Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft '17, pages 46–57. IEEE Press.

[Cruz and Abreu, 2018] Cruz, L. and Abreu, R. (2018). Using Automatic Refactoring to Improve Energy Efficiency of Android Apps. CoRR, abs/1803.05889.

[Cruz and Abreu, 2019a] Cruz, L. and Abreu, R. (2019a). Catalog of energy patterns for mobile applications. Empirical Software Engineering, 24(4):2209–2235.

[Cruz and Abreu, 2019b] Cruz, L. and Abreu, R. (2019b). On the energy footprint of mobile testing frameworks. IEEE Transactions on Software Engineering.

[Cruz et al., 2019] Cruz, L., Abreu, R., Grundy, J., Li, L., and Xia, X. (2019). Do energy-oriented changes hinder maintainability?

[Cruz et al., 2017] Cruz, L., Abreu, R., and Rouvignac, J.-N. (2017). Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. In IEEE/ACM International Conference on Mobile Software Engineering and Systems, MobileSoft 2017, MOBILESoft '17, pages 205–206.

[David et al., 2010] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. (2010). RAPL: Memory Power Estimation and Capping. In Proc. of 16th ACM/IEEE Int. Symposium on Low Power Electronics and Design, ISLPED '10, pages 189–194.

[Desrochers et al., 2016] Desrochers, S., Paradis, C., and Weaver, V. M. (2016). A validation of dram rapl power measurements. In Proceedings of the Second International Symposium on Memory Systems, MEMSYS '16, page 455–470, New York, NY, USA. Association for Computing Machinery.

[Di Nucci et al., 2017a] Di Nucci, D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., and De Lucia, A. (2017a). Petra: A software-based tool for estimating the energy profile of android applications. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 3–6.

[Di Nucci et al., 2017b] Di Nucci, D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., and De Lucia, A. (2017b). Software-based energy profiling of android apps: Simple, efficient and reliable? In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 103–114.

[Fahad et al., 2019] Fahad, M., Shahid, A., Manumachu, R., and Lastovetsky, A. (2019). A comparative study of methods for measurement of energy of computing. Energies, 12.

[Feitosa et al., 2017] Feitosa, D., Alders, R., Ampatzoglou, A., Avgeriou, P., and Nakagawa, E. (2017). Investigating the effect of design patterns on energy consumption. Journal of software-Evolution and process, 29(2 Special issue).

[Gamma et al., 1994] Gamma, E., Vlissides, J., Johnson, R., and Helm, R. (1994). Design Patterns: Elements of reusable object-oriented software. Addison-Wesley Professional; 1 edition.

[Grech et al., 2014] Grech, N., Georgiou, K., Pallister, J., Kerrison, S., and Eder, K. (2014). Static energy consumption analysis of LLVM IR programs. CoRR, abs/1405.4565.

[Gui et al., 2015] Gui, J., Mcilroy, S., Nagappan, M., and Halfond, W. G. J. (2015). Truth in advertising: The hidden cost of mobile ads for software developers. In Proceedings of the 37th International Conference on Software Engineering (ICSE).

[Gustafsson et al., 2006] Gustafsson, J., Ermedahl, A., Sandberg, C., and Lisper, B. (2006). Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In 2006 27th IEEE Int. Real-Time Systems Symposium (RTSS'06), pages 57–66.

[Hähnel et al., 2012] Hähnel, M., Döbel, B., Völp, M., and Härtig, H. (2012). Measuring Energy Consumption for Short Code Paths Using RAPL. SIGMETRICS Perform. Eval. Rev., 40(3):13–17.

[Hao et al., 2012] Hao, S., Li, D., Halfond, W., and Govindan, R. (2012). Estimating Android applications' CPU energy usage via bytecode profiling. In Green and Sustainable Software (GREENS), 2012 First Int. Workshop on, pages 1–7.

[Hao et al., 2013] Hao, S., Li, D., Halfond, W. G. J., and Govindan, R. (2013). Estimating mobile application energy consumption using program analysis. In Proc. of 35th Int. Conf. on Software Engineering (ICSE).

[Harmon and Auseklis, 2009] Harmon, R. R. and Auseklis, N. (2009). Sustainable IT services: Assessing the impact of green computing practices. In Management of Engineering & Technology, 2009. PICMET 2009. Portland Int. Conf. on, pages 1707–1717. IEEE.

[Hasan et al., 2016] Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., and Hindle, A. (2016). Energy Profiles of Java Collections Classes. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pages 225–236, New York, NY, USA. ACM.

[Hegde et al., 2016] Hegde, N., Melanson, E. L., and Sazonov, E. (2016). Development of a real time activity monitoring Android application utilizing SmartStep. In Proceedings of the 2016 IEEE 38th Annual International Conference of the Engineering in Medicine and Biology Society (EMBC), pages 1886–1889.

[Hindle et al., 2014] Hindle, A., Wilson, A., Rasmussen, K., Barlow, E. J., Campbell, J. C., and Romansky, S. (2014). Greenminer: A hardware based mining software repositories software energy consumption framework. In Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, page 12–21. Association for Computing Machinery.

[Hooper, 2008] Hooper, A. (2008). Green computing. Communication of the ACM, 51(10):11–13.

[Hoque et al., 2015] Hoque, M. A., Siekkinen, M., Khan, K. N., Xiao, Y., and Tarkoma, S. (2015). Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices. ACM Comput. Surv., 48(3):39:1–39:40.

[Horcas et al., 2019] Horcas, J.-M., Pinto, M., and Fuentes, L. (2019). Context-aware energy-efficient applications for cyber-physical systems. Ad Hoc Networks, 82:15 – 30.

[Hu et al., 2018] Hu, Y., Yan, J., Yan, D., Lu, Q., and Yan, J. (2018). Lightweight energy consumption analysis and prediction for Android applications. Science of Computer Programming, pages 132–147. Special Issue on TASE 2016.

[Jabbarvand et al., 2016] Jabbarvand, R., Sadeghi, A., Bagheri, H., and Malek, S. (2016). Energy-aware test-suite minimization for android apps. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, page 425–436. Association for Computing Machinery.

[Jabbarvand et al., 2015] Jabbarvand, R., Sadeghi, A., Garcia, J., Malek, S., and Ammann, P. (2015). EcoDroid: An Approach for Energy-based Ranking of Android Apps. In Proc. of 4th Int. Workshop on Green and Sustainable Software, GREENS '15, pages 8–14. IEEE Press.

[Jiang et al., 2017] Jiang, H., Yang, H., Qin, S., Su, Z., Zhang, J., and Yan, J. (2017). Detecting Energy Bugs in Android Apps Using Static Analysis. In Duan, Z. and Ong, L., editors, Formal Methods and Software Engineering, pages 192–208. Springer International Publishing.

[Jung et al., 2012] Jung, W., Kang, C., Yoon, C., Kim, D., and Cha, H. (2012). DevScope: A Nonintrusive and Online Power Analysis Tool for Smartphone Hardware Components. In Proc. of Eighth IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis, CODES+ISSS '12, pages 353–362. ACM.

[Khalid et al., 2015] Khalid, H., Shihab, E., Nagappan, M., and Hassan, A. E. (2015). What Do Mobile App Users Complain About? IEEE Software, 32(3):70–77.

[Kirkpatrick et al., 1983] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. Science, 220(4598):671–680.

[Kjærgaard and Blunck, 2012] Kjærgaard, M. and Blunck, H. (2012). Unsupervised Power Profiling for Mobile Devices. In Puiatti, A. and Gu, T., editors, Mobile and Ubiquitous Systems: Computing, Networking, and Services, volume 104 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 138–149. Springer Berlin Heidelberg.

[Koomey, 2011] Koomey, J. (2011). Growth in data center electricity use 2005 to 2010. A report by Analytical Press, completed at the request of The New York Times.

[Li and Halfond, 2014] Li, D. and Halfond, W. G. J. (2014). An Investigation into Energy-saving Programming Practices for Android Smartphone App Development. In Proc. of 3rd Int. Workshop on Green and Sustainable Software, GREENS 2014, pages 46–53. ACM.

[Li et al., 2014a] Li, D., Hao, S., Gui, J., and Halfond, W. G. (2014a). An empirical study of the energy consumption of android applications. In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME).

[Li et al., 2013] Li, D., Hao, S., Halfond, W. G. J., and Govindan, R. (2013). Calculating Source Line Level Energy Information for Android Applications. In Proc. of 2013 Int. Symposium on Software Testing and Analysis, ISSTA 2013, pages 78–89. ACM.

[Li et al., 2014b] Li, D., Jin, Y., Sahin, C., Clause, J., and Halfond, W. G. J. (2014b). Integrated Energy-directed Test Suite Optimization. In Proc. of 2014 Int. Symposium on Software Testing and Analysis, ISSTA 2014, pages 339–350. ACM.

[Li et al., 2016] Li, D., Lyu, Y., Gui, J., and Halfond, W. G. J. (2016). Automated Energy Optimization of HTTP Requests for Mobile Applications. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pages 249–260. ACM.

[Li and Malik, 1995] Li, Y. S. and Malik, S. (1995). Performance Analysis of Embedded Software Using Implicit Path Enumeration. In Proc. of 32Nd Annual ACM/IEEE Design Automation Conference, DAC '95, pages 456–461. ACM.

[Li et al., 2015] Li, Z., Avgeriou, P., and Liang, P. (2015). A systematic mapping study on technical debt and its management. J. Syst. Softw., 101(C):193–220.

[Lima et al., 2016] Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., and Fernandes, J. P. (2016). Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In 2016 IEEE 23rd Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 517–528.

[Linares-Vásquez et al., 2014] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D. (2014). Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In Proc. of 11th Working Conf. on Mining Software Repositories, MSR 2014, pages 2–11. ACM.

[Liqat et al., 2014] Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M. V., and Eder, K. (2014). Energy Consumption Analysis of Programs Based on XMOS ISA-Level Models. In Gupta, G. and Peña, R., editors, Logic-Based Program Synthesis and Transformation, volume 8901 of LNCS, pages 72–90. Springer Int. Publishing.

[Liu et al., 2015] Liu, K., Pinto, G., and Liu, Y. D. (2015). Data-Oriented Characterization of Application-Level Energy Optimization. In Egyed, A. and Schaefer, I., editors, Fundamental Approaches to Software Engineering, volume 9033 of LNCS, pages 316–331. Springer Berlin Heidelberg.

[Lopes et al., 2010] Lopes, C., Bajracharya, S., Ossher, J., and Baldi, P. (2010). UCI Source Code Data Sets.

[Lopes et al., 2017] Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., and Vitek, J. (2017). DéJàVu: A Map of Code Duplicates on GitHub. Proc. ACM Program. Lang., 1(OOPSLA):84:1–84:28.

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

[Manotas et al., 2014] Manotas, I., Pollock, L., and Clause, J. (2014). SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. In Proc. of 36th Int. Conf. on Software Engineering, ICSE 2014, pages 503–514. ACM.

[Masanet et al., 2020] Masanet, E., Shehabi, A., Lei, N., Smith, S., and Koomey, J. (2020). Recalibrating global data center energy-use estimates. Science, 367(6481):984–986.

[Matalonga et al., 2019] Matalonga, H., Cabral, B., Castor, F., Couto, M., Pereira, R., Melo de Sousa, S., and Fernandes, J. P. (2019). GreenHub Farmer: Real-world data for Android Energy Mining. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pages 1–5. (to appear).

[Medeiros et al., 2016] Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., and Apel, S. (2016). A comparison of 10 sampling algorithms for configurable systems. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, page 643–654. Association for Computing Machinery.

[Melfe et al., 2018] Melfe, G., Fonseca, A., and Fernandes, J. a. P. (2018). Evaluation of the impact on energy consumption of lazy versus strict evaluation of haskell data-structures. In Proceedings of the XXII Brazilian Symposium on Programming Languages, SBLP '18, page 83–89. Association for Computing Machinery.

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

[Møller and Schwartzbach, 2015] Møller, A. and Schwartzbach, M. I. (2015). Static program analysis. Department of Computer Science, Aarhus University.

[Morales et al., 2018] Morales, R., Saborido, R., Khomh, F., Chicano, F., and Antoniol, G. (2018). EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. IEEE Transactions on Software Engineering, 44(12):1176–1206.

[Nakajima, 2013] Nakajima, S. (2013). Model-based Power Consumption Analysis of Smartphone Applications. In 16th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, September 29TH, 2013.

[Nakajima, 2015a] Nakajima, S. (2015a). Model Checking of Energy Consumption Behavior. In Cardin, M.-A., Krob, D., Lui, P. C., Tan, Y. H., and Wood, K., editors, Complex Systems Design & Management Asia, pages 3–14. Springer Int. Publishing.

[Nakajima, 2015b] Nakajima, S. (2015b). Using Real-Time Maude to Model Check Energy Consumption Behavior. In Bjørner, N. and de Boer, F., editors, FM 2015: Formal Methods, volume 9109 of LNCS, pages 378–394. Springer Int. Publishing.

[Palomba et al., 2017] Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., and De Lucia, A. (2017). Lightweight detection of Android-specific code smells: Tshe aDoctor project. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 487–491.

[Palomba et al., 2019] Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., and De Lucia, A. (2019). On the impact of code smells on the energy consumption of mobile applications. Information and Software Technology, 105:43–55.

[Pandruvada, 2014] Pandruvada, S. (2014). Intel® power governor. `https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl`. Accessed: October 19, 2020.

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

[Pathak et al., 2012] Pathak, A., Hu, Y. C., and Zhang, M. (2012). Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In Proc. of 7th ACM European Conf. on Computer Systems, EuroSys '12, pages 29–42. ACM.

[Pathak et al., 2011] Pathak, A., Hu, Y. C., Zhang, M., Bahl, P., and Wang, Y. (2011). Fine-grained Power Modeling for Smartphones Using System Call Tracing. In Proc. of Sixth Conf. on Computer Systems, EuroSys '11, pages 153–168. ACM.

[Pereira, 2018] Pereira, R. (2018). Energyware Engineering: Techniques and Tools for Green Software Development. PhD thesis, Universidade do Minho.

[Pereira et al., 2017a] Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017a). Helping Programmers Improve the Energy Efficiency of Source Code. In Proc. of the 39th International Conference on Soft. Eng. Companion, ICSE-C 2017, pages 238–240. ACM.

[Pereira et al., 2016] Pereira, R., Couto, M., Cunha, J., Fernandes, J. P., and Saraiva, J. (2016). The Influence of the Java Collection Framework on Overall Energy Consumption. In Proc. of 5th Int. Workshop on Green and Sustainable Software, GREENS '16, pages 15–21. ACM.

[Pereira et al., 2017b] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017b). 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. ACM.

[Peterson et al., 2011] Peterson, P., Singh, D., Kaiser, W., and Reiher, P. (2011). Investigating energy and security trade-offs in the classroom with the atom leap testbed. In 4th Workshop on Cyber Security Experimentation and Test, CSET 2011 ; Conference date: 08-08-2011.

[Pinto and Castor, 2014] Pinto, G. and Castor, F. (2014). Characterizing the energy efficiency of java's thread-safe collections in a multi-core environment. In Proc. of SPLASH'2014 workshop on Software Engineering for Parallel Systems (SEPS), SEPS, volume 14.

[Pinto and Castor, 2017] Pinto, G. and Castor, F. (2017). Energy efficiency: A new concern for application software developers. Commun. ACM, 60(12):68–75.

[Pinto et al., 2014] Pinto, G., Castor, F., and Liu, Y. D. (2014). Mining Questions About Software Energy Consumption. In Proc. of 11th Working Conf. on Mining Software Repositories, MSR 2014, pages 22–31. ACM.

[Pronk, 1999] Pronk, B. J. (1999). Medical Product Line Architectures, pages 357–367. Springer US.

[Rua et al., 2019a] Rua, R., Couto, M., Pinto, A., Cunha, J., and Saraiva, J. (2019a). Towards using Memoization for Saving Energy in Android. In Proceedings of the XXII Iberoamerican Conference on Software Engineering, CIbSE, pages 279–292.

[Rua et al., 2019b] Rua, R., Couto, M., and Saraiva, J. a. (2019b). GreenSource: A Large-scale Collection of Android Code, Tests and Energy Metrics. In Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19, pages 176–180, Piscataway, NJ, USA. IEEE Press.

[Saborido et al., 2018] Saborido, R., Morales, R., Khomh, F., Guéhéneuc, Y.-G., and Antoniol, G. (2018). Getting the most from map data structures in Android. Empirical Software Engineering, 23(5):2829–2864.

[Sahin et al., 2012] Sahin, C., Cayci, F., Gutierrez, I. L. M., Clause, J., Kiamilev, F., Pollock, L., and Winbladh, K. (2012). Initial explorations on design pattern energy usage. In Green and Sustainable Software (GREENS), 2012 First Int. Workshop on, pages 55–61. IEEE.

[Sahin et al., 2014a] Sahin, C., Pollock, L., and Clause, J. (2014a). How Do Code Refactorings Affect Energy Usage? In Proc. of 8th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement, ESEM '14, pages 36:1–36:10. ACM.

[Sahin et al., 2014b] Sahin, C., Tornquist, P., Mckenna, R., Pearso, Z., and Clause, J. (2014b). How Does Code Obfuscation Impact Energy Usage? In Proc. of 2014 IEEE Int. Conf. on Software Maintenance and Evolution, ICSME '14, pages 131–140. IEEE Computer Society.

[Sarkar et al., 2015] Sarkar, A., Guo, J., Siegmund, N., Apel, S., and Czarnecki, K. (2015). Cost-efficient sampling for performance prediction of configurable systems. In Proc. of 30th Int. Conf. on Automated Soft. Eng., ASE '15, pages 342–352. IEEE Comp. Soc.

[Sawilowsky, 2009] Sawilowsky, S. (2009). New Effect Size Rules of Thumb. Journal of Modern Applied Statistical Methods, 8:597–599.

[Sharp, 1998] Sharp, D. C. (1998). Reducing avionics software cost through component based product line development. In 17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference. Proceedings (Cat. No.98CH36267), volume 2, pages G32/1–G32/8 vol.2.

[Siegmund et al., 2015] Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. (2015). Performance-influence models for highly configurable systems. In Proc. of 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 284–294. ACM.

[Siegmund et al., 2013] Siegmund, N., von Rhein, A., and Apel, S. (2013). Family-based performance measurement. In Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences, GPCE '13, pages 95–104. ACM.

[Steger et al., 2004] Steger, M., Tischer, C., Boss, B., Müller, A., Pertler, O., Stolz, W., and Ferber, S. (2004). Introducing pla at bosch gasoline systems: Experiences and practices. In Nord, R. L., editor, Software Product Lines, pages 34–50. Springer Berlin Heidelberg.

[Tsintzira et al., 2019] Tsintzira, A., Ampatzoglou, A., Matei, O., Ampatzoglou, A., Chatzigeorgiou, A., and Heb, R. (2019). Technical debt quantification through metrics: An industrial validation. In 15th China-Europe International Symposium on Software Engineering Education (CEISEE' 19), pages –. IEEE.

[van Ommering, 2002] van Ommering, R. (2002). Building product populations with software components. In Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pages 255–265. ACM.

[Varshosaz et al., 2018] Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M. R., and Schaefer, I. (2018). A classification of product sampling for software product lines. In Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC '18, page 1–13. Association for Computing Machinery.

[Vekris et al., 2012] Vekris, P., Jhala, R., Lerner, S., and Agarwal, Y. (2012). Towards Verifying Android Apps for the Absence of No-sleep Energy Bugs. In Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems, HotPower'12. USENIX Association.

[Venkata et al., 2009] Venkata, S. K., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., and Taylor, M. B. (2009). SD-VBS: The San Diego Vision Benchmark Suite. In Proc. of

2009 IEEE Int. Symposium on Workload Characterization (IISWC), IISWC '09, pages 55–64. IEEE Computer Society.

[Verdecchia et al., 2018] Verdecchia, R., Saez, R. A., Procaccianti, G., and Lago, P. (2018). Empirical Evaluation of the Energy Impact of Refactoring Code Smells. In Penzenstadler, B., Easterbrook, S., Venters, C., and Ahmed, S. I., editors, ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability, volume 52 of EPiC Series in Computing, pages 365–383.

[Wilhelm et al., 2008] Wilhelm, R., Engblom, J., Ermedahl, A., TransHolsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. (2008). The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. ACM Trans. Embed. Comput. Syst., 7(3):36:1–36:53.

[Yoon et al., 2012] Yoon, C., Kim, D., Jung, W., Kang, C., and Cha, H. (2012). AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring. In Presented as part of the 2012 USENIX Annual Technical Conf. (USENIX ATC 12), pages 387–400. USENIX.

[Zhang et al., 2010] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. (2010). Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In Proc. of Eighth Int. Conf. on Hardware/Software Codesign and System Synthesis, CODES/ISSS '10, pages 105–114. ACM.

8 Chapter 3 Appendix

A.1 Full Study Results

Table A.1: Study Results for San Diego Benchmark SPL

Product	Observed (J)	Estimated (J)	Difference (J)	% Error
#1	114.170	114.75	0.5800	0.51%
#2	90.800	100.73	9.9300	10.94%
#3	87.930	100.72	12.7900	14.55%
#4	64.910	66.54	1.6300	2.51%
#5	64.470	70.08	5.6100	8.70%
#6	58.870	64.93	6.0600	10.29%
#7	56.800	67.06	10.2600	18.06%

Table A.2: Study Results for Elevator SPL

Product	Observed (J)	Estimated (J)	Difference (J)	% Error
#1	0.0139	0.0141	0.0002	1.34%
#2	0.0146	0.0155	0.0010	6.56%
#3	0.0149	0.0166	0.0017	11.24%
#4	0.0149	0.0149	0.0000	0.12%
#5	0.0148	0.0167	0.0019	12.57%
#6	0.0148	0.0151	0.0003	1.93%
#7	0.0155	0.0156	0.0001	0.50%
#8	0.0151	0.0152	0.0001	0.82%
#9	0.0149	0.0157	0.0008	5.55%
#10	0.0150	0.0178	0.0028	18.28%
#11	0.0149	0.0183	0.0034	22.96%
#12	0.0151	0.0186	0.0035	23.09%
#13	0.0150	0.0191	0.0041	27.45%
#14	0.0150	0.0177	0.0027	17.82%
#15	0.0151	0.0181	0.0030	20.20%
#16	0.0156	0.0185	0.0029	18.69%
#17	0.0149	0.0190	0.0041	27.28%
#18	0.0149	0.0154	0.0005	3.47%
#19	0.0157	0.0158	0.0001	0.88%
#20	0.0150	0.0156	0.0006	3.96%

Table A.3: Study Results for EmailSystem SPL

Product	Observed (J)	Estimated (J)	Difference (J)	% Error
#1	1.4832	2.4707	0.9875	66.58%
#2	1.4769	2.6124	1.1355	76.89%
#3	1.4818	2.4769	0.9950	67.15%
#4	1.4472	2.6185	1.1713	80.93%
#5	1.4865	2.4883	1.0018	67.40%
#6	1.4342	2.6299	1.1958	83.38%
#7	1.4750	2.4944	1.0195	69.12%
#8	1.4436	2.6361	1.1925	82.60%
#9	1.4427	2.4707	1.0280	71.25%
#10	1.4172	2.6124	1.1952	84.33%
#11	1.4637	3.1394	1.6758	114.49%
#12	1.4511	3.2811	1.8300	126.11%
#13	1.3988	3.1395	1.7407	124.44%
#14	1.4977	3.2811	1.7835	119.08%
#15	1.4399	3.8082	2.3683	164.48%
#16	1.4006	3.9498	2.5492	182.00%
#17	1.4702	2.4769	1.0067	68.47%
#18	1.4112	2.6185	1.2074	85.56%
#19	1.4541	3.1456	1.6914	116.32%
#20	1.4462	3.2872	1.8410	127.29%
#21	1.4343	3.1456	1.7113	119.31%
#22	1.4565	3.2873	1.8308	125.70%
#23	1.4816	3.8143	2.3328	157.45%
#24	1.4397	3.9560	2.5163	174.78%
#25	1.4232	2.4883	1.0651	74.84%
#26	1.4423	2.6299	1.1876	82.34%
#27	1.4775	3.1570	1.6795	113.67%
#28	1.4180	3.2986	1.8806	132.63%
#29	1.4935	3.1570	1.6635	111.38%
#30	1.4769	3.2987	1.8218	123.36%
#31	1.4473	3.8258	2.3784	164.33%
#32	1.4283	3.9674	2.5391	177.76%
#33	1.4957	2.4944	0.9987	66.77%
#34	1.4513	2.6361	1.1848	81.64%
#35	1.4303	3.1631	1.7329	121.16%
#36	1.4438	3.3048	1.8610	128.90%
#37	1.4910	3.1632	1.6722	112.16%
#38	1.4036	3.3048	1.9012	135.45%
#39	1.4287	3.8319	2.4032	168.20%
#40	1.4435	3.9735	2.5300	175.27%

Table A.4: Study Results for Minepump SPL

Product	Observed (J)	Estimated (J)	Difference (J)	% Error
#1	0.1340	0.1464	0.0123	9.19%
#2	0.1336	0.1562	0.0225	16.87%
#3	0.1348	0.1566	0.0217	16.11%
#4	0.1355	0.1550	0.0195	14.42%
#5	0.1346	0.1468	0.0121	9.02%
#6	0.1348	0.1473	0.0124	9.22%
#7	0.1364	0.1664	0.0300	21.97%
#8	0.1452	0.1648	0.0196	13.53%
#9	0.1396	0.1566	0.0170	12.18%
#10	0.1371	0.1571	0.0200	14.58%
#11	0.1374	0.1652	0.0278	20.19%
#12	0.1380	0.1570	0.0190	13.78%
#13	0.1411	0.1574	0.0164	11.61%
#14	0.1388	0.1554	0.0166	11.94%
#15	0.1395	0.1559	0.0164	11.73%
#16	0.1356	0.1477	0.0121	8.91%
#17	0.1344	0.1750	0.0406	30.24%
#18	0.1369	0.1668	0.0299	21.86%
#19	0.1361	0.1673	0.0312	22.92%
#20	0.1405	0.1652	0.0247	17.59%
#21	0.1366	0.1657	0.0291	21.31%
#22	0.1389	0.1575	0.0186	13.38%
#23	0.1381	0.1656	0.0276	19.96%
#24	0.1347	0.1661	0.0313	23.26%
#25	0.1342	0.1579	0.0236	17.60%
#26	0.1339	0.1563	0.0224	16.72%
#27	0.1401	0.1754	0.0353	25.19%
#28	0.1356	0.1759	0.0403	29.73%
#29	0.1354	0.1677	0.0323	23.82%
#30	0.1346	0.1661	0.0316	23.46%
#31	0.1364	0.1665	0.0301	22.11%
#32	0.1348	0.1763	0.0415	30.81%

Chapter 4 Appendix

B

B.1 Energy Consumption Distribution: Before and After Refactoring EGAPs

For each EGAP or combination of EGAPs, we present a boxplot comparing the consumptions before and after refactoring. The values considered on each plot are respective to all tests executed on all applications where the corresponding EGAP/combination was refactored.

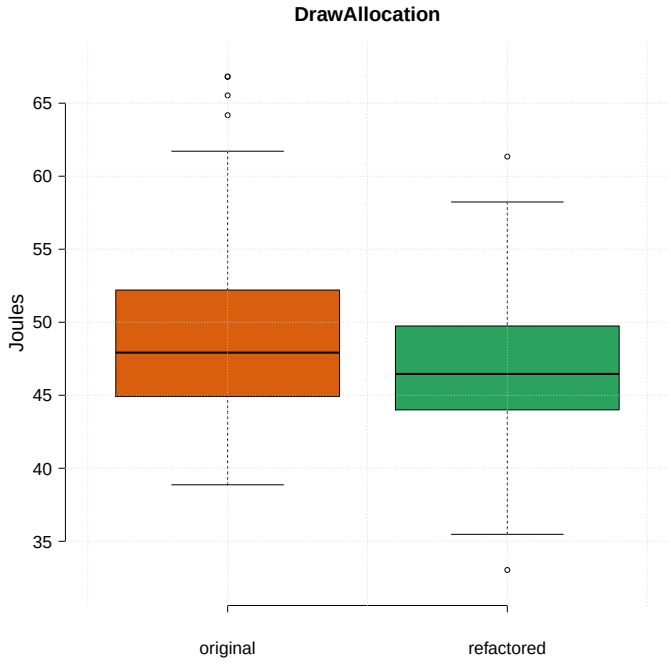


Figure B.1: Energy Before/After Refactoring.
EGAPs: `DrawAllocation` (DA)

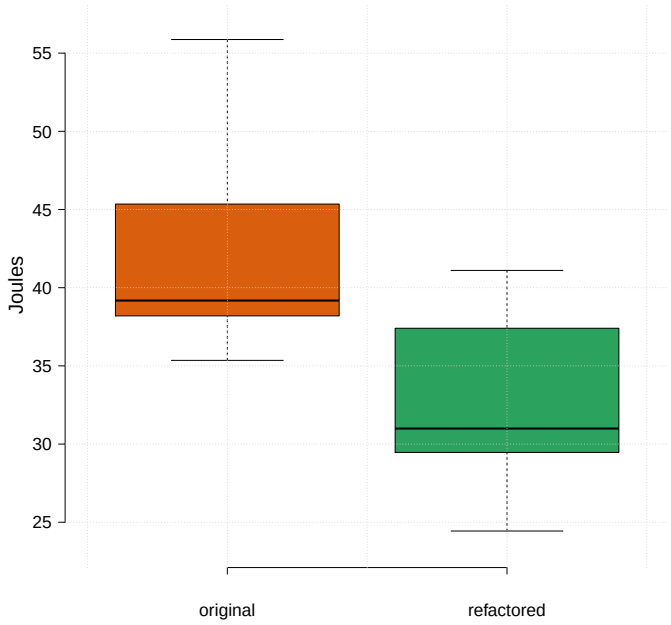


Figure B.2: Energy Before/After Refactoring.
EGAPs: `Wakelock` (`WL`)

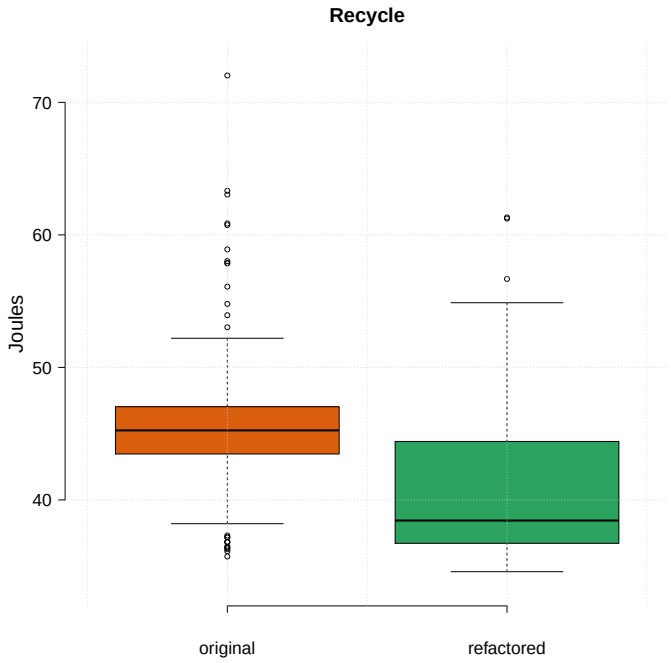


Figure B.3: Energy Before/After Refactoring.
EGAPs: `Recyle` (`R`)

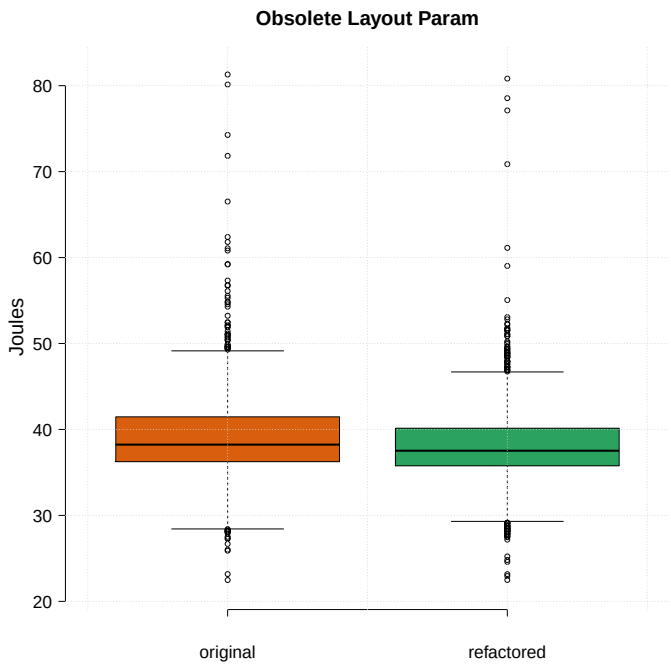


Figure B.4: Energy Before/After Refactoring.
EGAPs: `ObsoleteLayoutParam` (OLP)

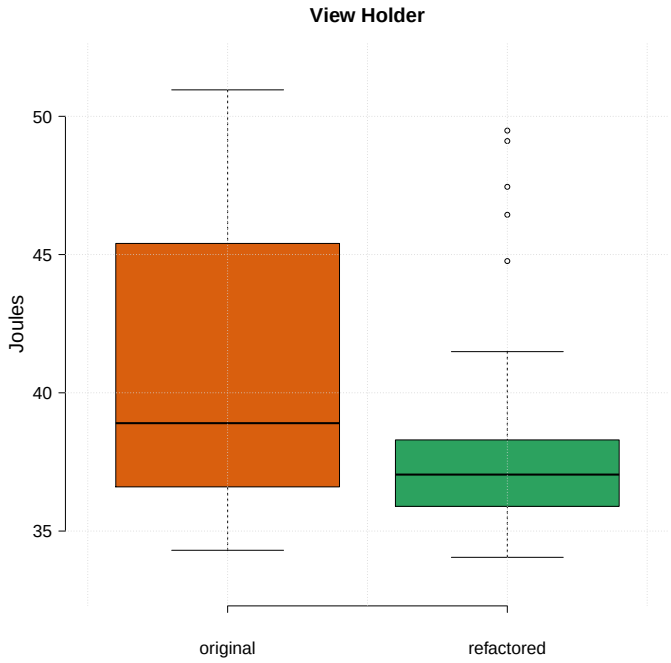


Figure B.5: Energy Before/After Refactoring.
EGAPs: `ViewHolder` (VH)

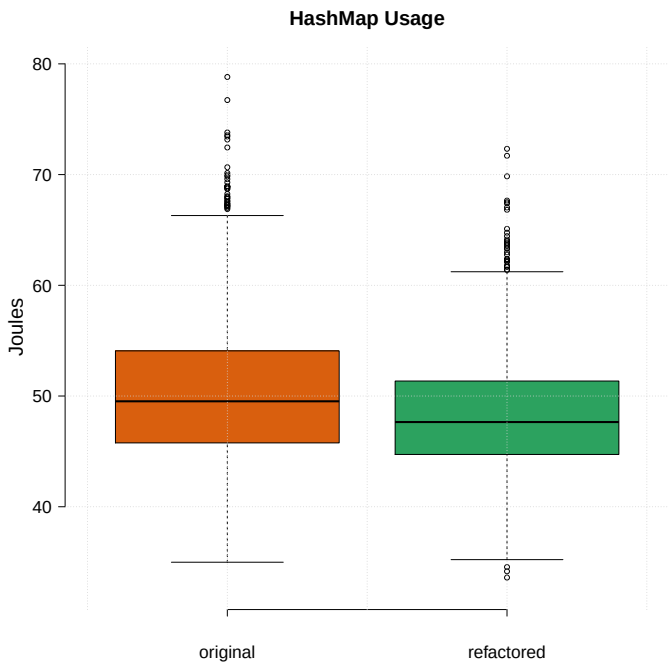


Figure B.6: Energy Before/After Refactoring.
EGAPs: `HashMapUsage` (`HMU`)

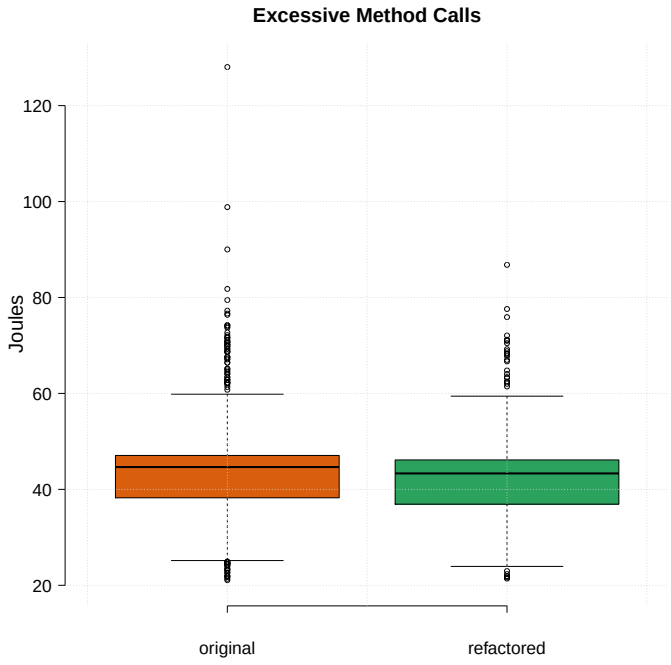


Figure B.7: Energy Before/After Refactoring.
EGAPs: `ExcessiveMethodCalls` (`EMC`)

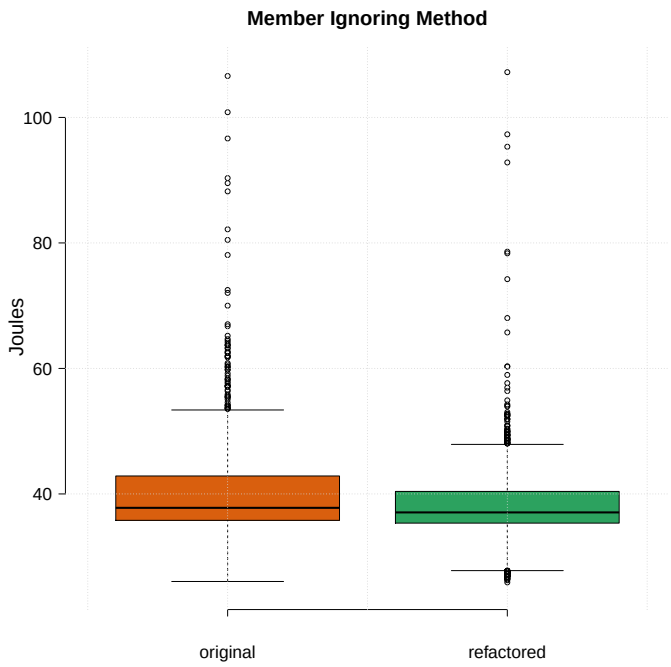


Figure B.8: Energy Before/After Refactoring.
EGAPs: `MemberIgnoringMethod` (`MIM`)

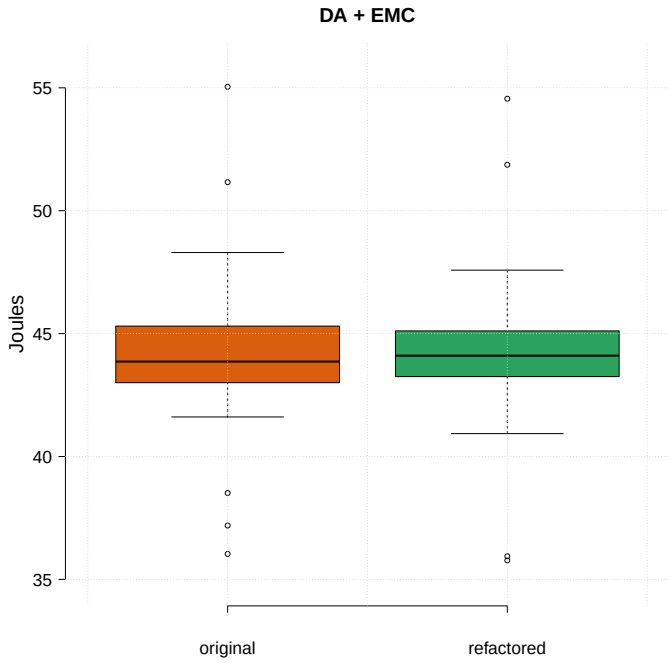


Figure B.9: Energy Before/After Refactoring.
EGAPs: `DA`, `EMC`

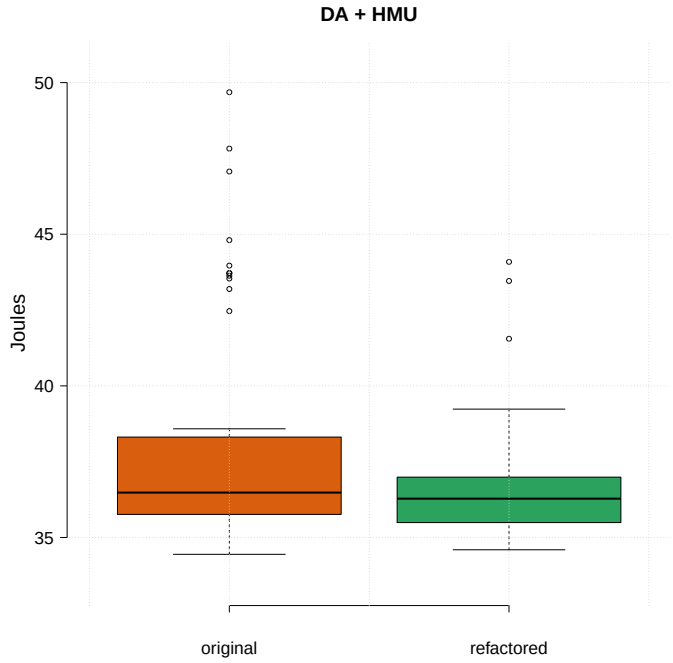


Figure B.10: Energy Before/After Refactoring.
EGAPs: `DA`, `HMU`

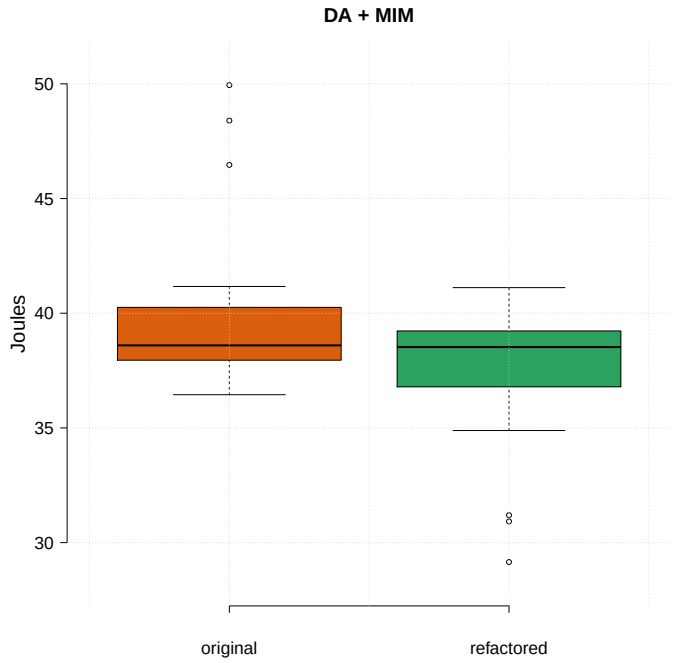


Figure B.11: Energy Before/After Refactoring.
EGAPs: `DA`, `MIM`

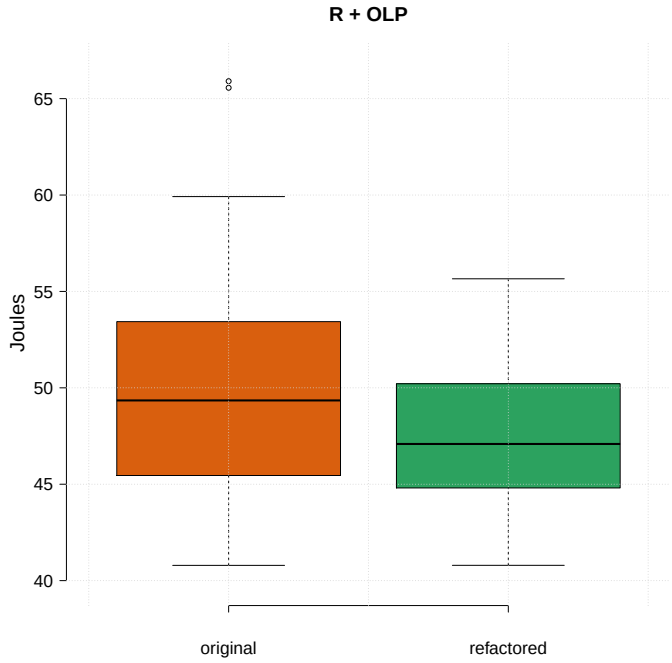


Figure B.12: Energy Before/After Refactoring.
EGAPs: R, OLP

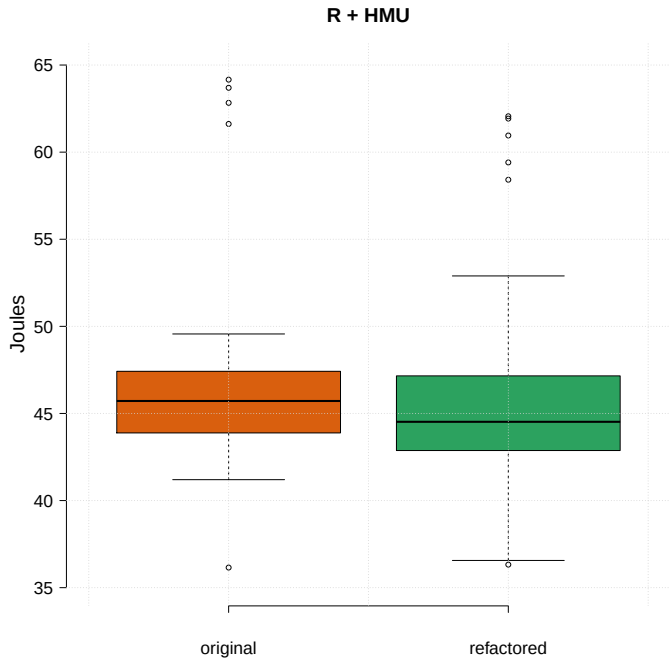


Figure B.13: Energy Before/After Refactoring.
EGAPs: R, HMU

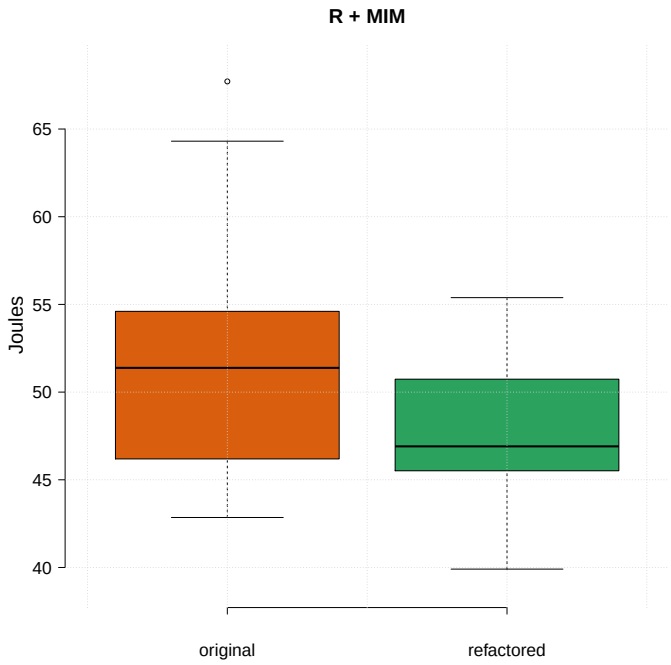


Figure B.14: Energy Before/After Refactoring.
EGAPs: `R`, `MIM`

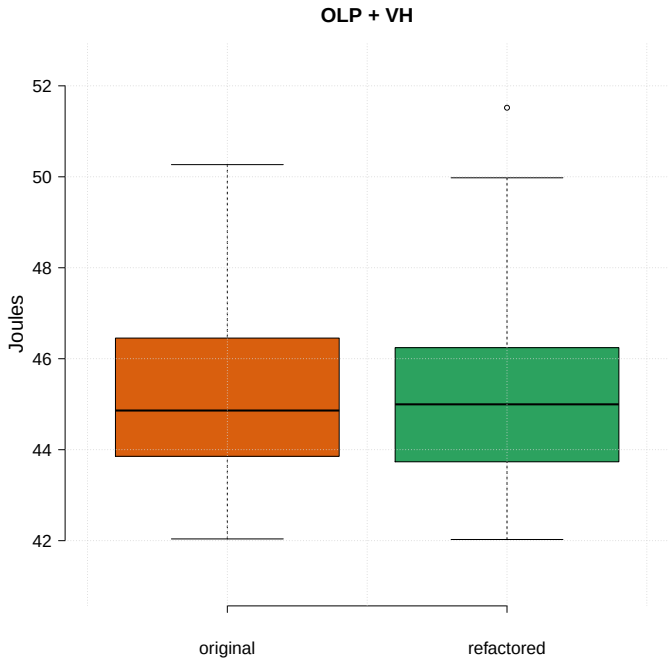


Figure B.15: Energy Before/After Refactoring.
EGAPs: `OLP`, `VH`

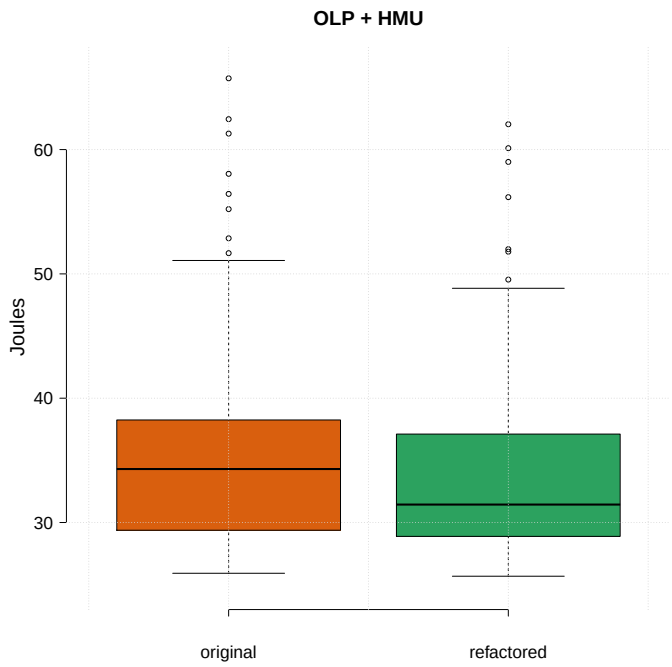


Figure B.16: Energy Before/After Refactoring.
EGAPs: `OLP`, `HMU`

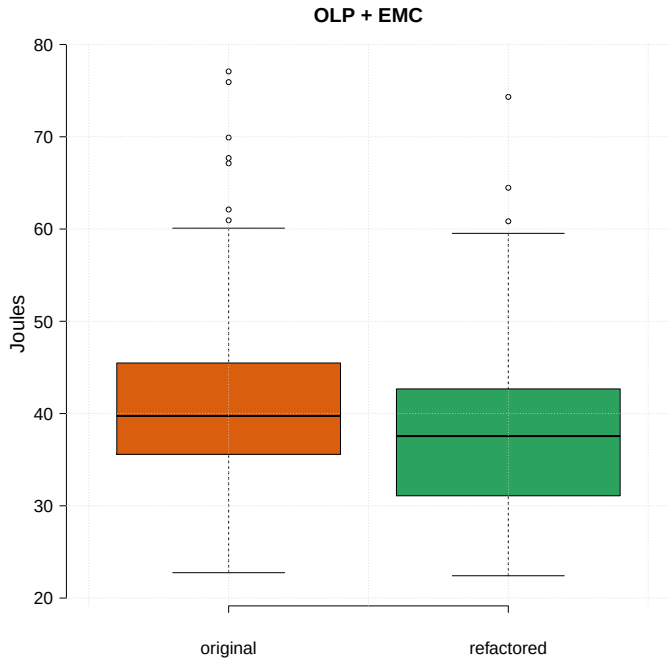


Figure B.17: Energy Before/After Refactoring.
EGAPs: `OLP`, `EMC`

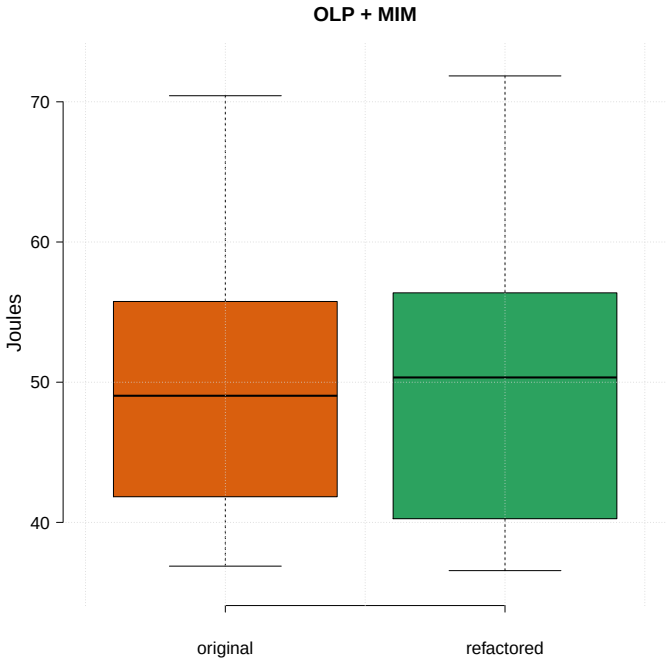


Figure B.18: Energy Before/After Refactoring.
EGAPs: `OLP`, `MIM`

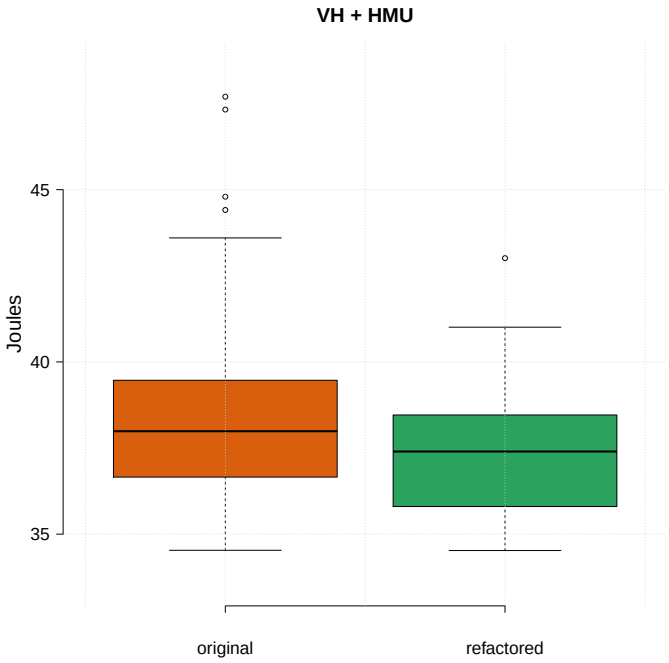


Figure B.19: Energy Before/After Refactoring.
EGAPs: `VH`, `HMU`

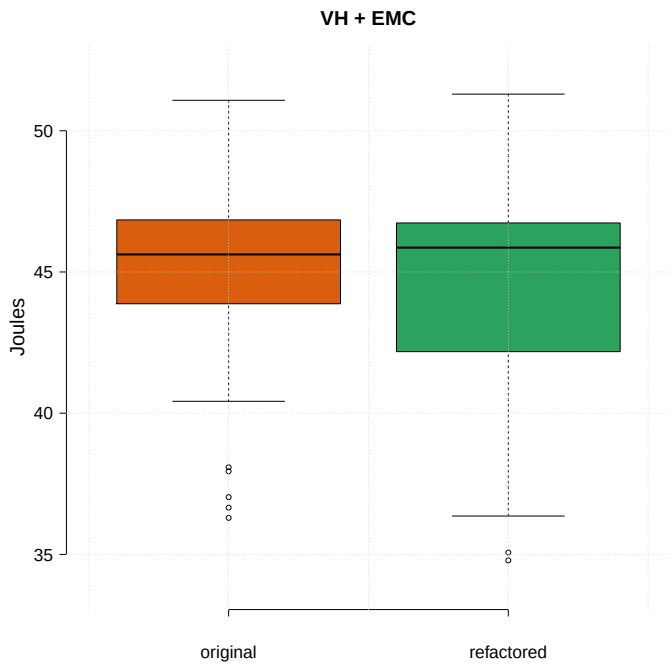


Figure B.20: Energy Before/After Refactoring.
EGAPs: `VH`, `EMC`

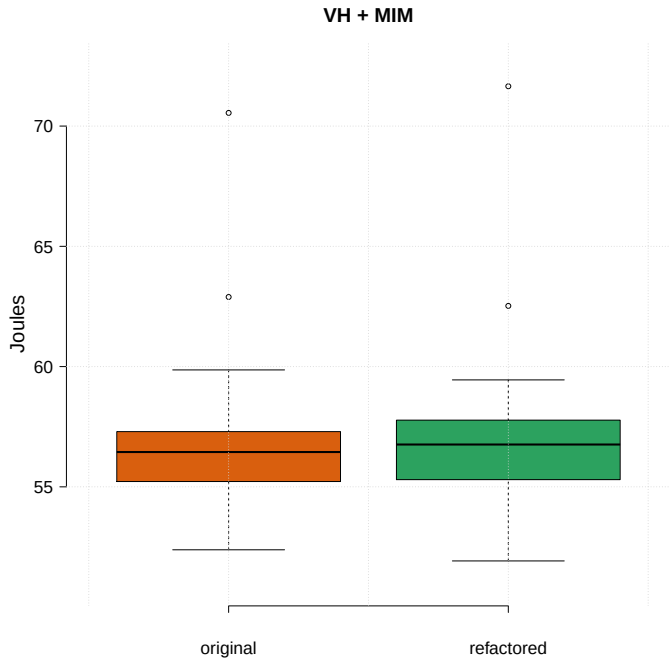


Figure B.21: Energy Before/After Refactoring.
EGAPs: `VH`, `MIM`

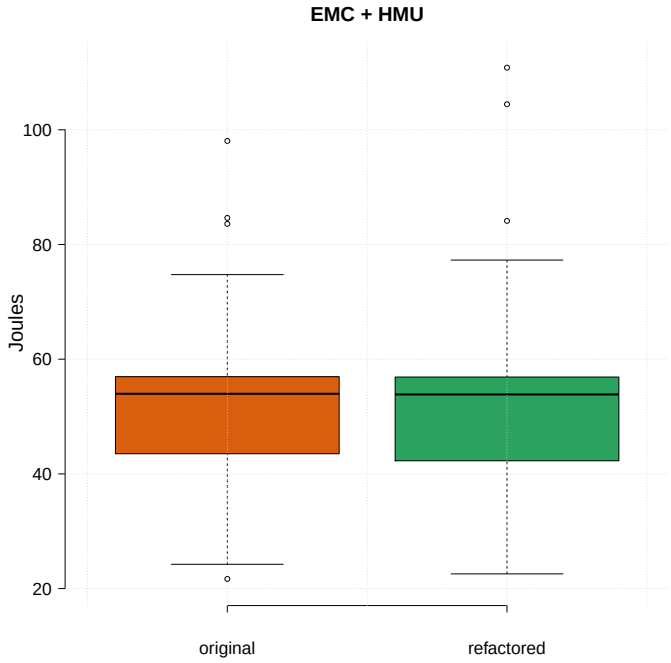


Figure B.22: Energy Before/After Refactoring.
EGAPs: `HMU`, `EMC`

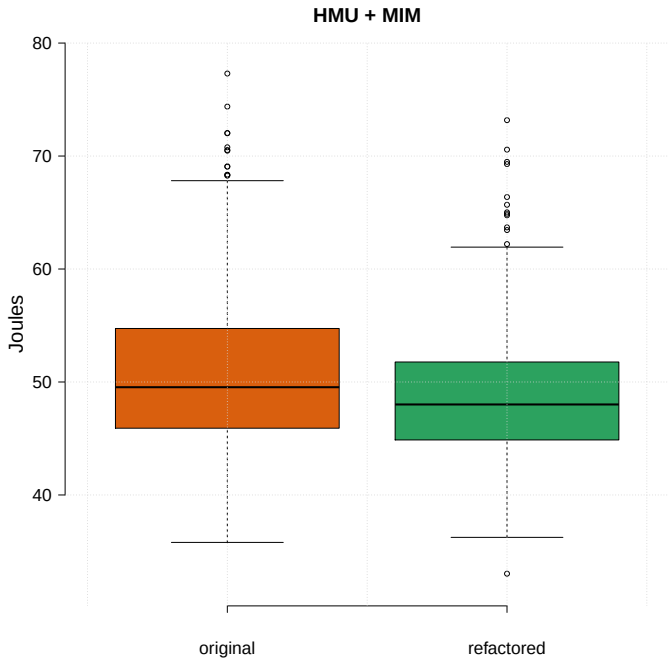


Figure B.23: Energy Before/After Refactoring.
EGAPs: `HMU`, `MIM`

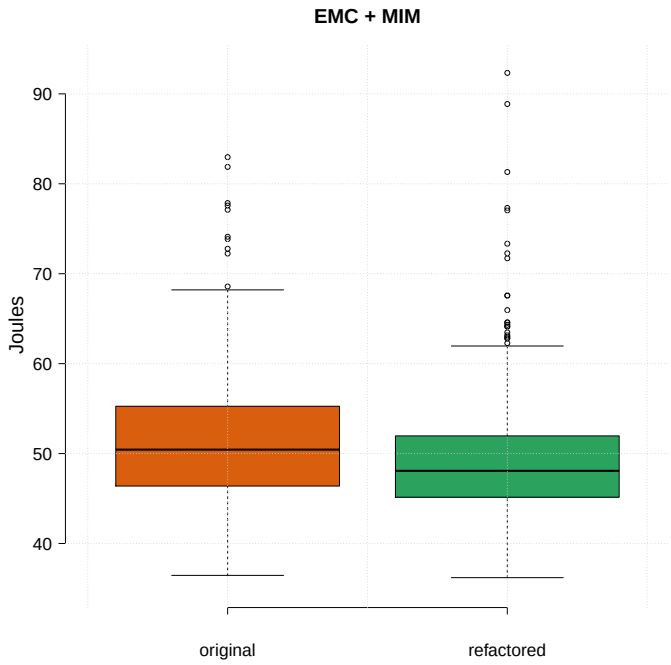


Figure B.24: Energy Before/After Refactoring.
EGAPs: `EMC`, `MIM`

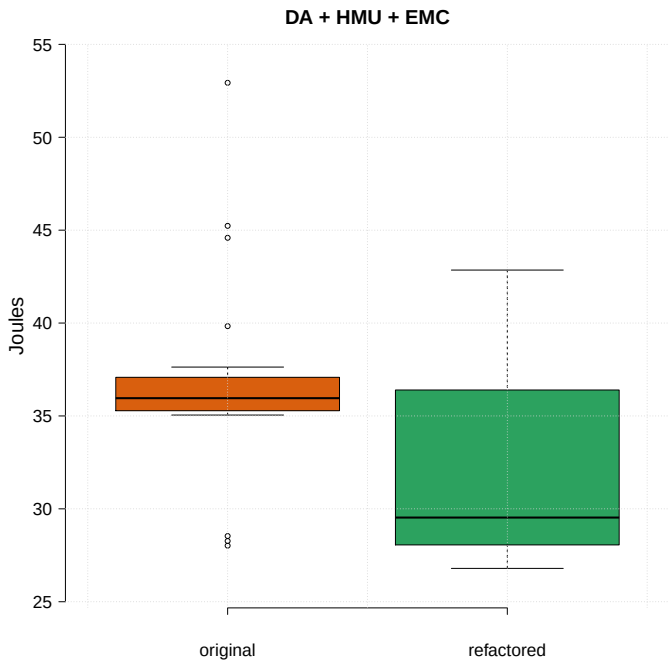


Figure B.25: Energy Before/After Refactoring.
EGAPs: `DA`, `HMU`, `EMC`

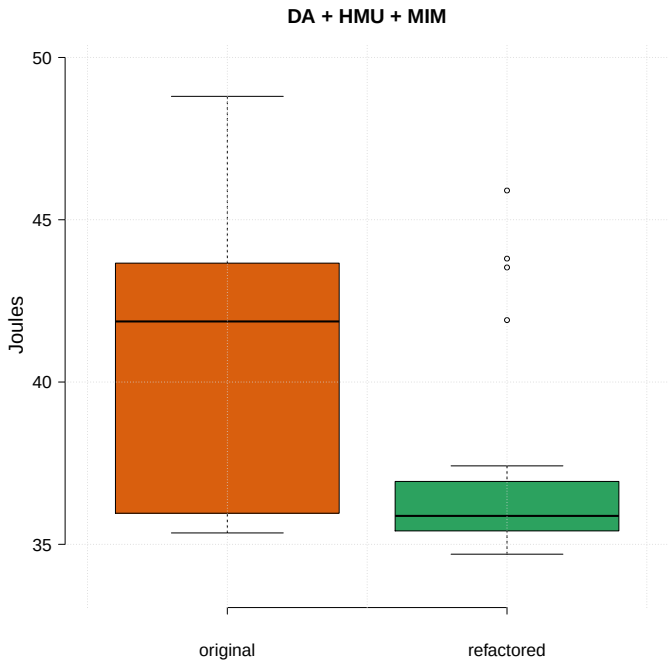


Figure B.26: Energy Before/After Refactoring.
EGAPs: `DA`, `HMU`, `MIM`

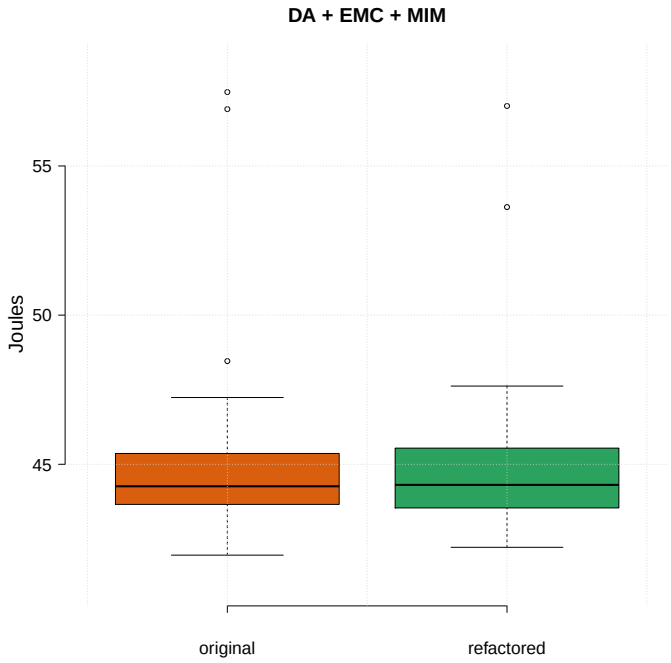


Figure B.27: Energy Before/After Refactoring.
EGAPs: `DA`, `EMC`, `MIM`

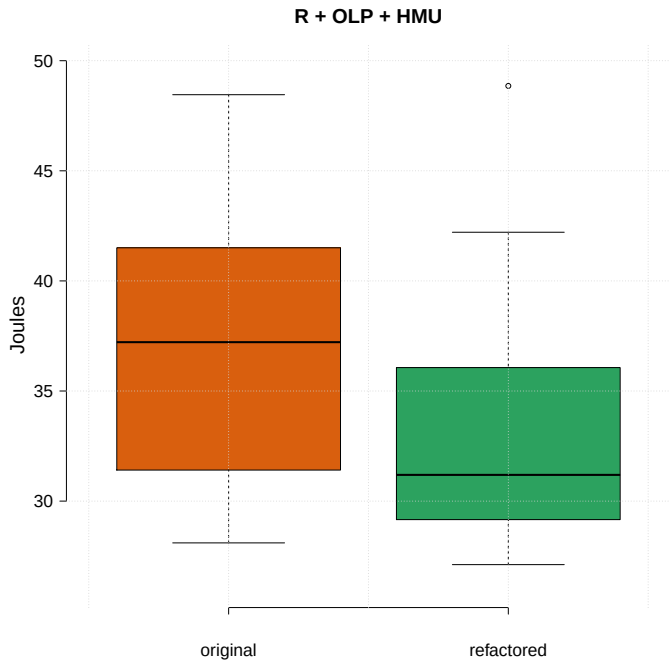


Figure B.28: Energy Before/After Refactoring.
EGAPs: `R`, `OLP`, `HMU`

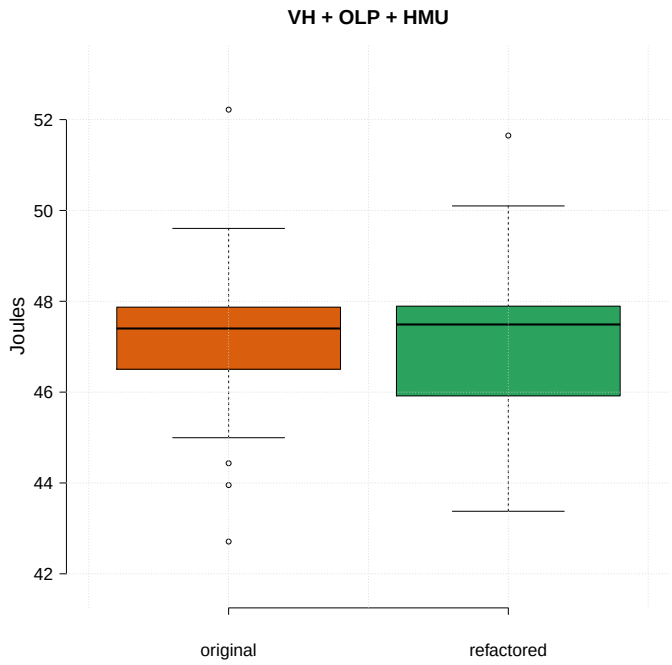


Figure B.29: Energy Before/After Refactoring.
EGAPs: `OLP`, `VH`, `HMU`

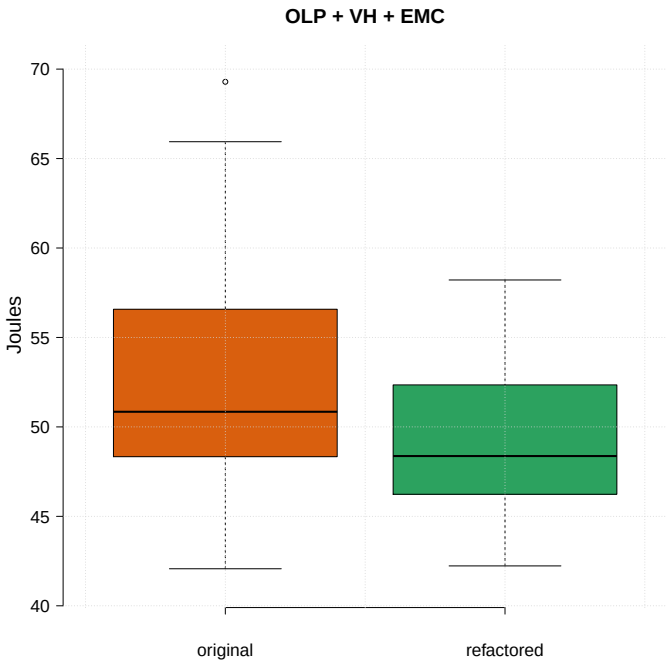


Figure B.30: Energy Before/After Refactoring.
EGAPs: `OLP`, `VH`, `EMC`

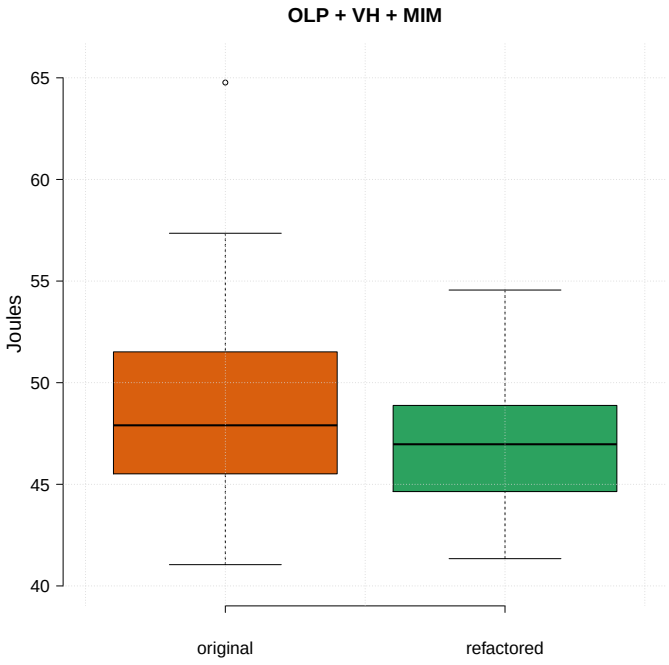


Figure B.31: Energy Before/After Refactoring.
EGAPs: `OLP`, `VH`, `MIM`

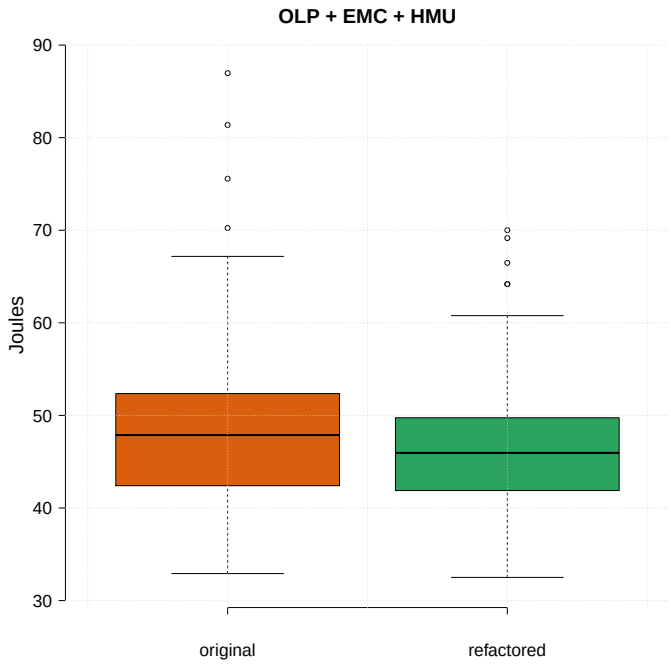


Figure B.32: Energy Before/After Refactoring.
EGAPs: `OLP`, `HMU`, `EMC`

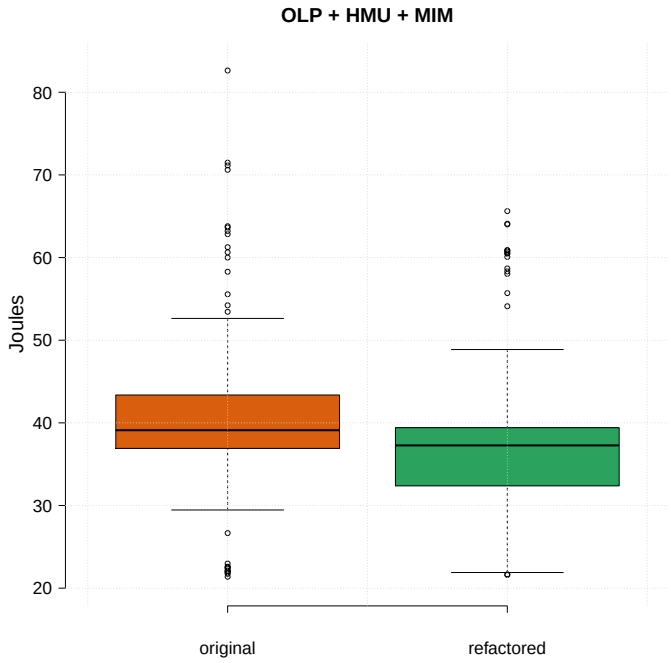


Figure B.33: Energy Before/After Refactoring.
EGAPs: `OLP`, `HMU`, `MIM`

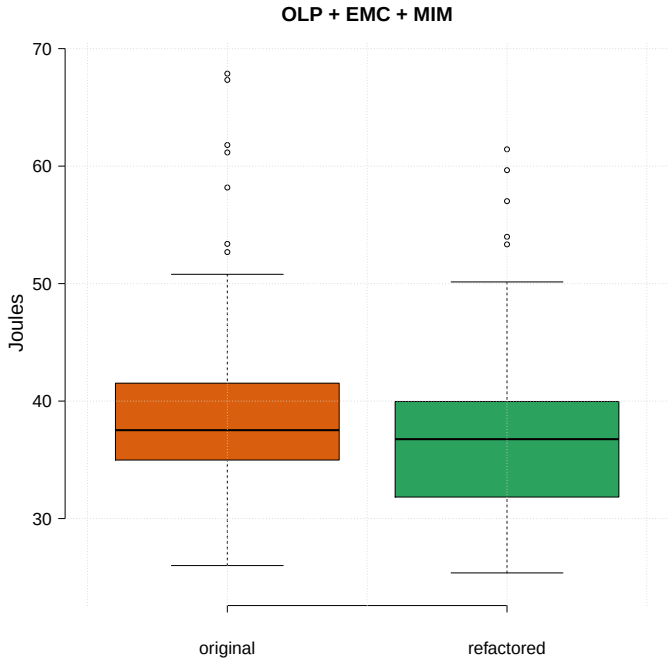


Figure B.34: Energy Before/After Refactoring.
EGAPs: `OLP`, `EMC`, `MIM`

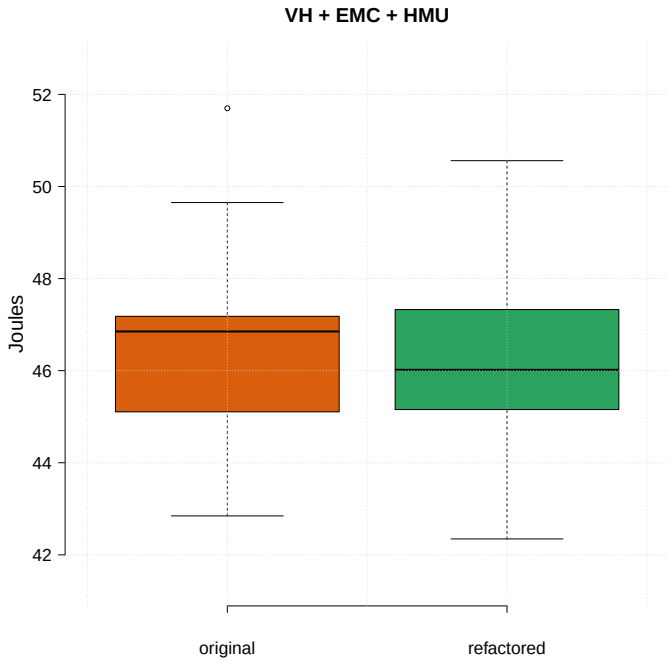


Figure B.35: Energy Before/After Refactoring.
EGAPs: `VH`, `HMU`, `EMC`

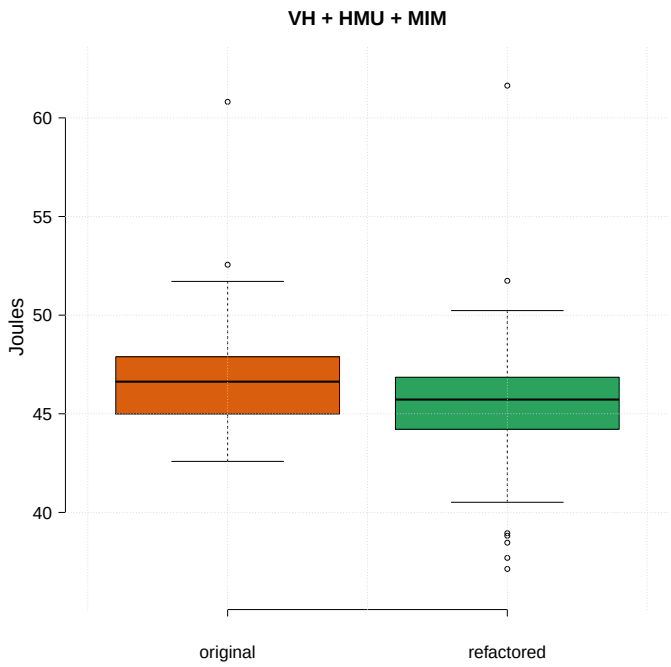


Figure B.36: Energy Before/After Refactoring.
EGAPs: `VH`, `HMU`, `MIM`

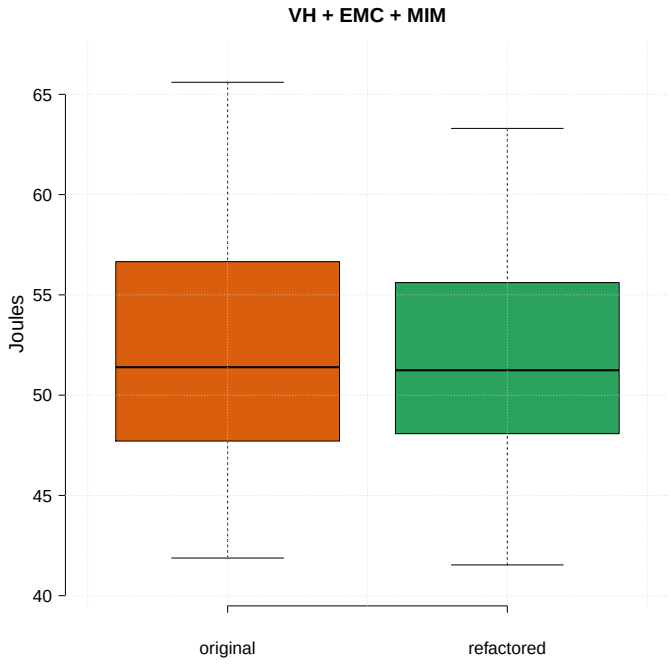


Figure B.37: Energy Before/After Refactoring.
EGAPs: `VH`, `EMC`, `MIM`

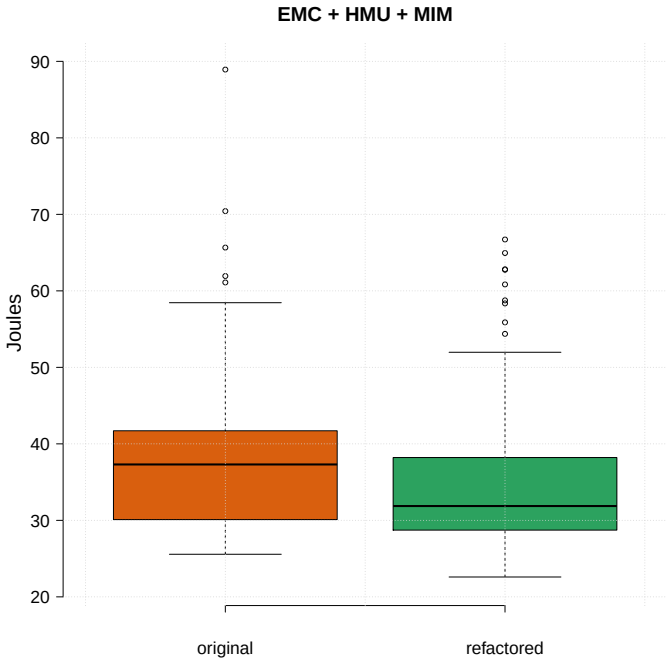


Figure B.38: Energy Before/After Refactoring.
EGAPs: `HMU`, `EMC`, `MIM`

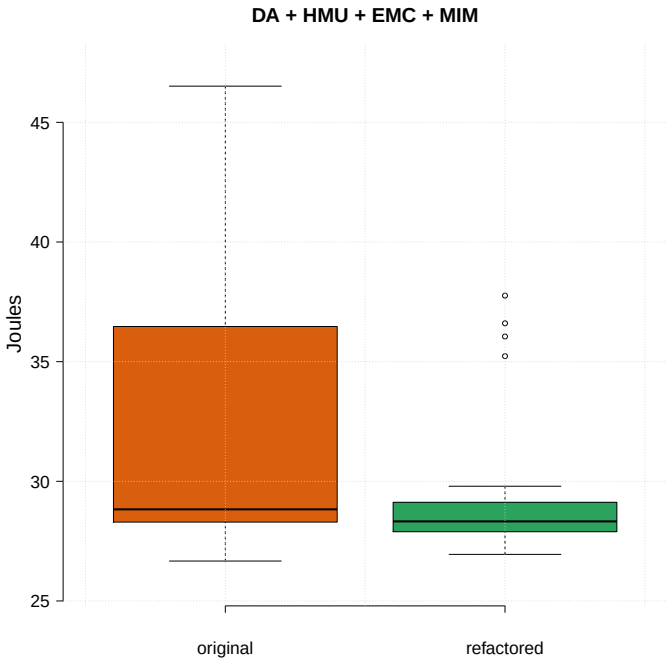


Figure B.39: Energy Before/After Refactoring.
EGAPs: `DA`, `EMC`, `HMU`, `MIM`

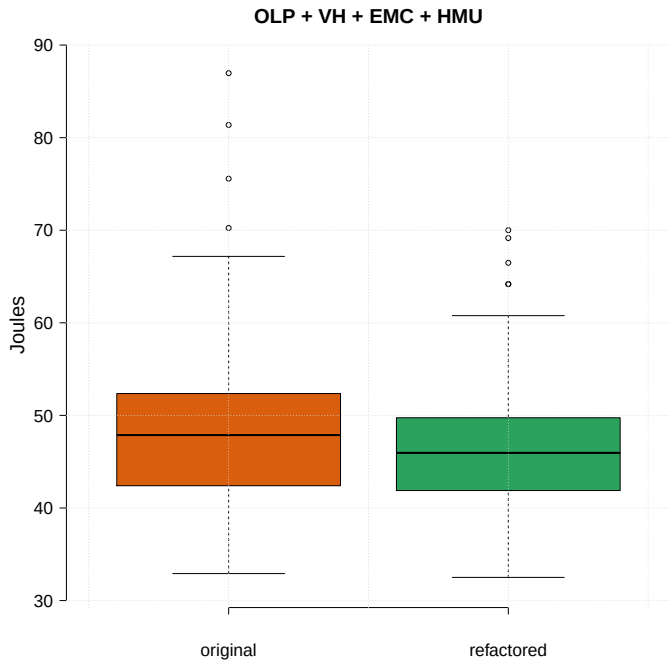


Figure B.40: Energy Before/After Refactoring.
EGAPs: `OLP`, `VH`, `HMU`, `EMC`

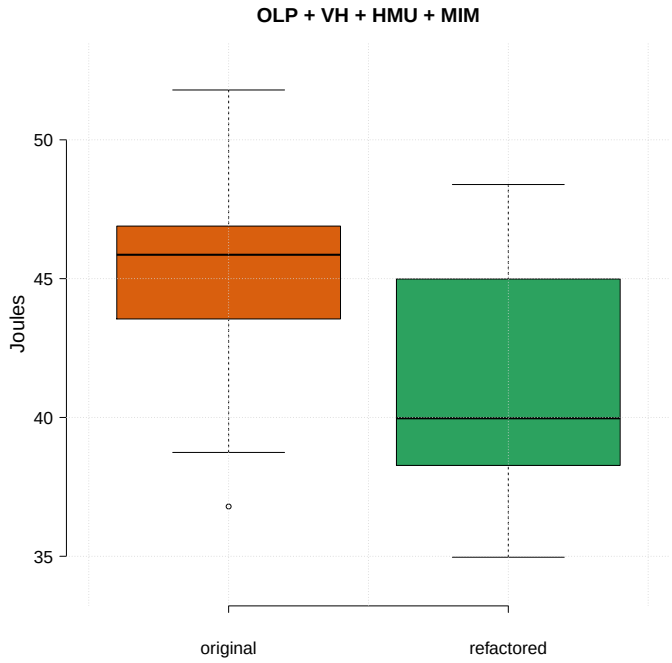


Figure B.41: Energy Before/After Refactoring.
EGAPs: `OLP`, `VH`, `HMU`, `MIM`

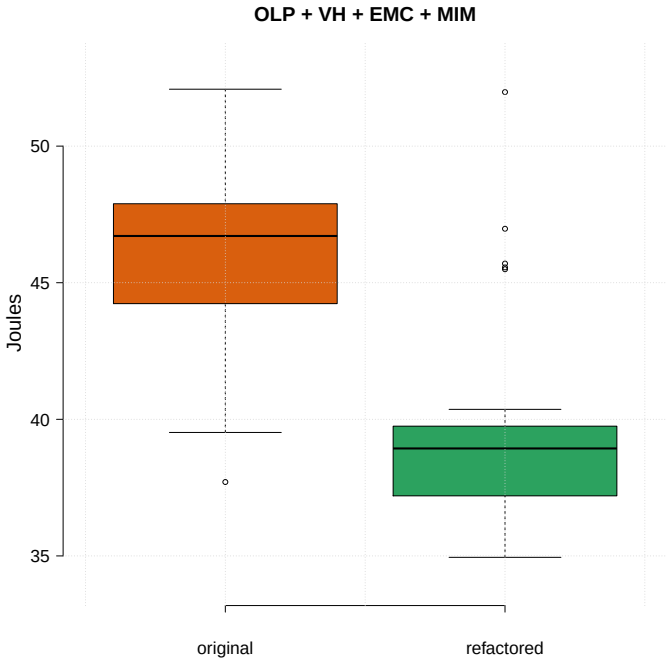


Figure B.42: Energy Before/After Refactoring.
EGAPs: `OLP`, `VH`, `EMC`, `MIM`

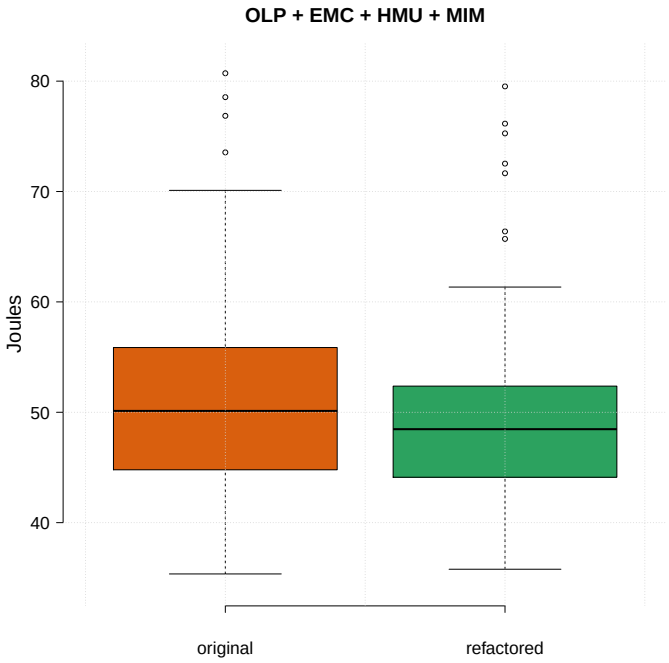


Figure B.43: Energy Before/After Refactoring.
EGAPs: `OLP`, `HMU`, `EMC`, `MIM`

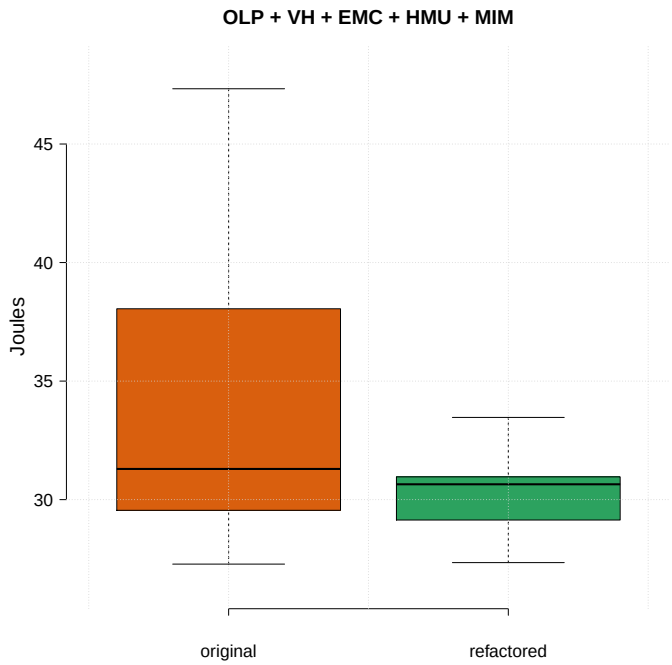


Figure B.44: Energy Before/After Refactoring.
EGAPs: `OLP`, `VH`, `HMU`, `EMC`, `MIM`

B.2 Mann–Whitney U and Cohen's d Results

In this section, we include the results of the Mann–Whitney U statistical test, to compare the gains of each individual EGAP e against the all combinations where e occurs. The Cohen's d coefficient is also included for the the scenarios where the test rejected the null hypothesis (highlighted in **bold**), meaning that the gains from are consistently higher/lower. This allows to quantify how big is the difference is.

Table B.1: Mann-Whitney U results: `DrawAllocation`

	Mann-Whitney U		**Cohen's d**
	p-value	stats	
DA **vs** DA + HMU	5.27E-02	-1.9372	–
DA **vs** DA + EMC	8.52E-01	-0.1861	–
DA **vs** DA + MIM	1.60E-01	-1.4067	–
DA **vs** DA + HMU + EMC	**8.76E-04**	-3.3276	1.3765
DA **vs** DA + HMU + MIM	**5.33E-04**	-3.4634	1.0798
DA **vs** DA + EMC + MIM	7.41E-01	-0.3309	–
DA **vs** DA + HMU + EMC + MIM	**2.38E-02**	-2.2604	0.8737

Table B.2: Mann-Whitney U results: `Recycle`

	Mann-Whitney U		**Cohen's d**
	p-value	stats	
R **vs** R + HMU	**4.76E-08**	5.4599	1.3511
R **vs** R + OLP	**1.17E-06**	4.8601	1.2503
R **vs** R + MIM	7.49E-01	0.3201	–
R **vs** R + OLP + HMU	8.25E-01	0.2206	–

Table B.3: Mann-Whitney U results: `ObsoleteLayoutParam`

	Mann-Whitney U		Cohen's d
	p-value	stats	
OLP **vs** OLP + HMU	2.13E-01	1.2458	–
OLP **vs** R + OLP	1.93E-01	-1.3029	–
OLP **vs** OLP + VH	9.07E-01	0.1172	–
OLP **vs** OLP + EMC	**3.81E-06**	-4.6213	0.3510
OLP **vs** OLP + MIM	7.19E-02	1.7999	–
OLP **vs** R + OLP + HMU	**8.73E-06**	-4.4465	1.0079
OLP **vs** OLP + VH + HMU	4.79E-01	0.7082	–
OLP **vs** OLP + HMU + EMC	5.08E-01	0.6621	–
OLP **vs** OLP + HMU + MIM	2.71E-01	-1.1004	–
OLP **vs** OLP + VH + EMC	3.18E-01	-0.9992	–
OLP **vs** OLP + VH + MIM	3.57E-01	-0.9216	–
OLP **vs** OLP + EMC + MIM	1.43E-01	-1.4650	–
OLP **vs** OLP + VH + HMU + EMC	**3.09E-03**	2.9589	0.9427
OLP **vs** OLP + VH + HMU + MIM	**4.89E-02**	-1.9694	0.7646
OLP **vs** OLP + HMU + EMC + MIM	8.26E-01	0.2199	–
OLP **vs** OLP + VH + EMC + MIM	**5.29E-07**	-5.0156	1.7327
OLP **vs** OLP + VH + HMU + EMC + MIM	**3.20E-02**	-2.1440	0.7024

Table B.4: Mann-Whitney U results: `ViewHolder`

	Mann-Whitney U		Cohen's d
	p-value	stats	
VH **vs** VH + HMU	2.76E-01	1.0892	–
VH **vs** OLP + VH	**4.13E-02**	2.0406	0.8554
VH **vs** VH + EMC	**9.13E-04**	3.3159	0.7353
VH **vs** VH + MIM	**1.34E-02**	2.4739	0.8372
VH **vs** OLP + VH + HMU	1.90E-01	1.3097	–
VH **vs** VH + HMU + EMC	6.77E-01	0.4172	–
VH **vs** VH + HMU + MIM	7.41E-01	-0.3309	–
VH **vs** OLP + VH + EMC	9.30E-01	-0.0873	–
VH **vs** OLP + VH + MIM	9.77E-01	-0.0291	–
VH **vs** VH + EMC + MIM	3.88E-01	0.8634	–
VH **vs** OLP + VH + HMU + EMC	**7.62E-04**	3.3664	1.1262
VH **vs** OLP + VH + HMU + MIM	4.43E-01	-0.7664	–
VH **vs** OLP + VH + EMC + MIM	**1.88E-04**	-3.7350	1.0371
VH **vs** OLP + VH + HMU + EMC + MIM	1.01E-01	-1.6395	–

Table B.5: Mann-Whitney U results: `HashMapUsage`

	Mann-Whitney U		Cohen's d
	p-value	stats	
HMU **vs** DA + HMU	8.63E-01	-0.1723	–
HMU **vs** OLP + HMU	7.54E-01	0.3128	–
HMU **vs** R + HMU	3.85E-01	0.8686	–
HMU **vs** VH + HMU	**1.16E-02**	2.5231	0.5095
HMU **vs** HMU + EMC	1.49E-01	1.4422	–
HMU **vs** HMU + MIM	8.93E-02	-1.6993	–
HMU **vs** DA + HMU + EMC	**1.66E-02**	-2.3963	0.9310
HMU **vs** DA + HMU + MIM	1.43E-01	-1.4649	–
HMU **vs** R + OLP + HMU	**3.46E-04**	-3.5779	0.6373
HMU **vs** OLP + VH + HMU	**4.89E-02**	1.9694	0.7039
HMU **vs** OLP + HMU + EMC	3.75E-01	0.8877	–
HMU **vs** OLP + HMU + MIM	**6.41E-03**	-2.7260	0.4611
HMU **vs** VH + HMU + EMC	2.73E-01	1.0963	–
HMU **vs** VH + HMU + MIM	1.52E-01	1.4339	–
HMU **vs** HMU + EMC + MIM	**6.23E-05**	-4.0038	0.5170
HMU **vs** DA + HMU + EMC + MIM	7.64E-01	-0.3007	–
HMU **vs** OLP + VH + HMU + EMC	**3.44E-04**	3.5798	1.0859
HMU **vs** OLP + VH + HMU + MIM	3.57E-01	-0.9216	–
HMU **vs** OLP + HMU + EMC + MIM	1.98E-01	1.2874	–
HMU **vs** OLP + VH + HMU + EMC + MIM	2.99E-01	-1.0381	–

Table B.6: Mann-Whitney U results: `ExcessiveMethodCalls`

	Mann-Whitney U		Cohen's d
	p-value	stats	
EMC **vs** DA + EMC	**1.34E-04**	3.8192	1.1011
EMC **vs** HMU + EMC	2.90E-01	1.0587	–
EMC **vs** OLP + EMC	**3.65E-06**	-4.6306	0.3579
EMC **vs** VH + EMC	7.76E-02	1.7648	–
EMC **vs** EMC + MIM	5.29E-01	-0.6303	–
EMC **vs** DA + HMU + EMC	5.60E-02	-1.9112	–
EMC **vs** DA + EMC + MIM	**1.72E-04**	3.7571	1.1276
EMC **vs** OLP + HMU + EMC	3.72E-01	-0.8930	–
EMC **vs** VH + HMU + EMC	5.54E-01	-0.5918	–
EMC **vs** HMU + EMC + MIM	**4.43E-06**	-4.5903	0.3751
EMC **vs** OLP + VH + EMC	3.37E-01	-0.9604	–
EMC **vs** OLP + EMC + MIM	7.01E-02	-1.8113	–
EMC **vs** VH + EMC + MIM	7.34E-01	0.3395	–
EMC **vs** DA + HMU + EMC + MIM	8.54E-01	0.1843	–
EMC **vs** OLP + VH + HMU + EMC	**3.29E-03**	2.9395	0.8751
EMC **vs** OLP + HMU + EMC + MIM	6.05E-02	-1.8775	–
EMC **vs** OLP + VH + EMC + MIM	**2.10E-07**	-5.1903	2.1278
EMC **vs** OLP + VH + HMU + EMC + MIM	**1.75E-02**	-2.3768	0.8542

Table B.7: Mann-Whitney U results: `MemberIgnoringMethod`

	Mann-Whitney U		Cohen's d
	p value	stats	
MIM **vs** DA + MIM	**4.76E-03**	2.8231	0.7073
MIM **vs** HMU + MIM	1.61E-01	-1.4028	–
MIM **vs** OLP + MIM	5.63E-01	0.5779	–
MIM **vs** R + MIM	**1.20E-02**	-2.5127	0.6529
MIM **vs** VH + MIM	**1.57E-02**	2.4157	0.6929
MIM **vs** EMC + MIM	1.86E-01	-1.3213	–
MIM **vs** DA + HMU + MIM	**1.07E-02**	-2.5515	1.0462
MIM **vs** DA + EMC + MIM	**8.07E-04**	3.3504	0.8754
MIM **vs** OLP + HMU + MIM	1.03E-01	-1.6308	–
MIM **vs** VH + HMU + MIM	8.52E-01	-0.1861	–
MIM **vs** HMU + EMC + MIM	**3.49E-06**	-4.6396	0.5323
MIM **vs** OLP + VH + MIM	4.79E-01	-0.7082	–
MIM **vs** OLP + EMC + MIM	8.96E-01	0.1303	–
MIM **vs** VH + EMC + MIM	6.91E-01	0.3978	–
MIM **vs** DA + HMU + EMC + MIM	2.11E-01	-1.2515	–
MIM **vs** OLP + VH + HMU + MIM	1.38E-01	-1.4843	–
MIM **vs** OLP + HMU + EMC + MIM	8.76E-02	1.7083	–
MIM **vs** OLP + VH + EMC + MIM	**3.07E-06**	-4.6664	1.3798
MIM **vs** OLP + VH + HMU + EMC + MIM	**4.67E-02**	-1.9888	0.5346