Report

# ENACT - ENergy efficiency through ArChitectural Tactics for Software Engineering

**Tareq Md Rabiul Hossain CHY**
Masters 1 (M1) in Computer Science (Cyber-Physical Social Systems)

Internship Supervisors : **Sophie CHABRIDON (1) and Denisse MUÑANTE ARZAPALO (2)**
(1) Institut Polytechnique de Paris / Télécom SudParis / SAMOVAR
(2) ENSIIE / SAMOVAR / Évry, France

MSc. Program Supervisor: **Maxime LEFRANCOIS (3)**
(3) Ecole des Mines de Saint-Étienne, Saint-Étienne, France

From 03/04/2023 to 31/08/2023

# Abstract

This internship report provides a comprehensive overview of the research conducted to explore tactics for improving software energy efficiency. The study was conducted under the supervision of Sophie Chabridon from Télécom SudParis/SAMOVAR Lab and Denisse Muñante Arzapalo from ENSIIE/ SAMOVAR Lab Évry, France. The research methodology involved a combination of experimental analysis and practical experiments.

The initial phase of the research involved an extensive literature review, which provided a solid foundation for the study. Various existing approaches, techniques, and technologies related to software energy efficiency were analyzed, enabling a deep understanding of the subject matter.

The internship experience was greatly beneficial, and I would like to express gratitude to the internship supervisors for their unwavering support throughout the entire process. Despite their busy schedules, the supervisors demonstrated attentiveness to my needs and facilitated a seamless integration into the team. Their guidance, advice, and assistance contributed significantly to the success of the internship, creating a positive and productive atmosphere.

Furthermore, I would like to extend appreciation to the Télécom SudParis/SAMOVAR Lab, E4C, for providing the opportunity to conduct the internship within their research environment. The collaborative and stimulating atmosphere of the lab enhanced the learning experience and fostered meaningful contributions to the field of software energy efficiency.

Additionally, I acknowledge the contributions of Professor Maxime Lefrançois, Professor Piere Maret, and all the teachers at Mines Saint-Etienne and Jean Monnet University. Their dedication and investment in the my academic pursuits have been instrumental in shaping the research skills and knowledge required for this internship. I express gratitude for their ongoing support and mentorship.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Introduction

Global warming is a significant environmental concern, and carbon emissions play a central role in its occurrence. These emissions, originating from various human activities, directly and indirectly contribute to the warming of the Earth's atmosphere. A primary contributor to global warming is the burning of fossil fuels, namely coal, natural gas, and oil, for the generation of electricity. It is crucial to note that fossil fuels are the largest contributors to global climate change, accounting for more than 75% of global greenhouse gas emissions and nearly 90% of all carbon dioxide emissions.[1]. Unfortunately, despite the availability of alternative energy sources, the majority of electricity production worldwide (almost two-thirds (63.3%) of global electricity)[2] continues to heavily rely on fossil fuels.

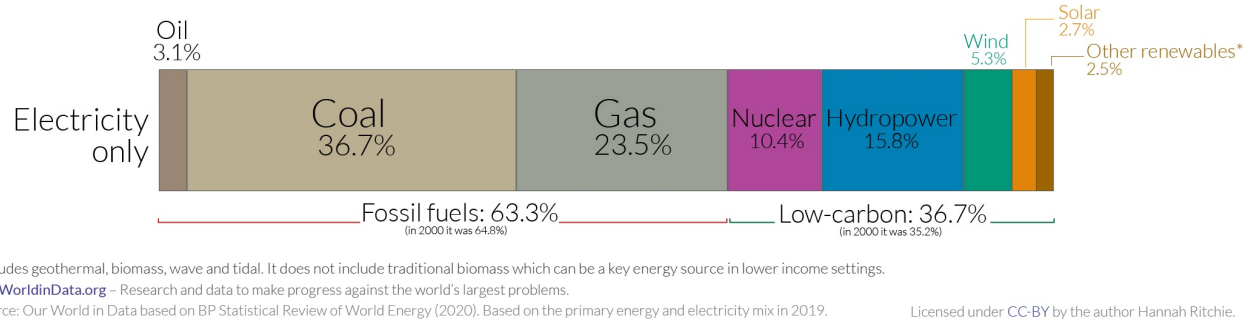


Figure 1.1: Global electricity production from fossil fuels[3].

The consumption of electricity has risen due to the increased use of devices in various sectors such as home, entertainment, and more. Moreover, the number of smart devices (IoT Devices) is expected to grow more than twice by the end of this decade. By 2040, billions of IoT devices could contribute up to 14 percent of the world's carbon emissions[4].

[1] https://www.un.org/en/climatechange/science/causes-effects-climate-change
[2] https://ourworldindata.org/electricity-mix
[3] https://ourworldindata.org/uploads/2020/08/Global-energy-vs.-electricity-breakdown-1536x812.pngx
[4] https://www.theguardian.com/environment/2017/dec/11/tsunami-of-data-could-consume-fifth-global-electricity-by-2025

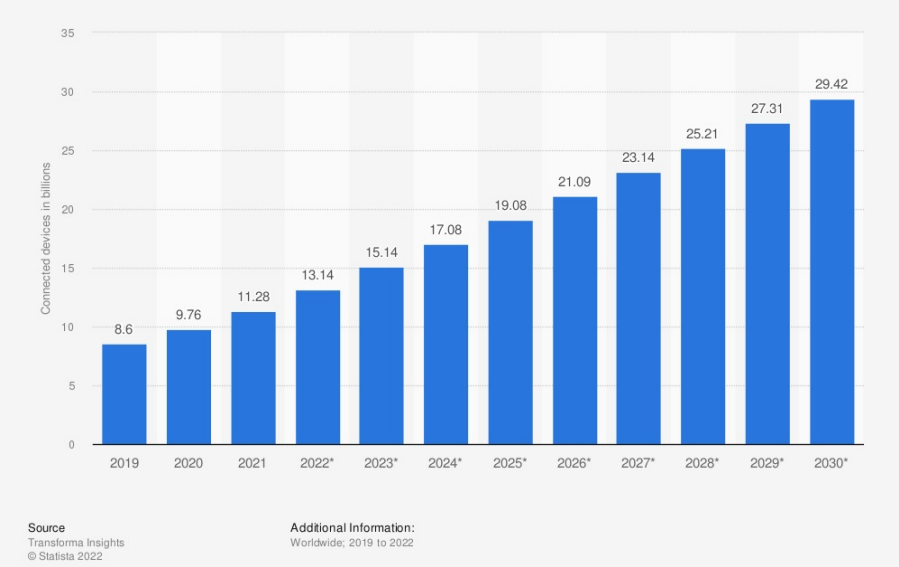


Figure 1.2: Number of IOT connected devices worldwide (2019-2021) with forecasts to 2030.[5]

As the use of IoT devices becomes more widespread, energy consumption has become a major concern. Researchers are working on ways to make both hardware and software components of devices more energy-efficient. While software itself does not consume energy, its architecture, structure, and usage context can influence the energy consumption of hardware. By configuring software to be more energy-efficient, we can reduce energy consumption and carbon emissions, contributing to the preservation of our planet.

## 1.2 Motivations

Global emissions of carbon dioxide ($CO_2$) have undergone significant changes over time. Before the Industrial Revolution, emissions were minimal. However, starting from the mid-20th century, emissions began to increase at a faster pace. In 1950, global $CO_2$ emissions were around 6 billion tonnes. By 1990, this figure nearly quadrupled, surpassing 22 billion tonnes. Currently, the world emits over 34 billion tonnes of $CO_2$ annually.[6]

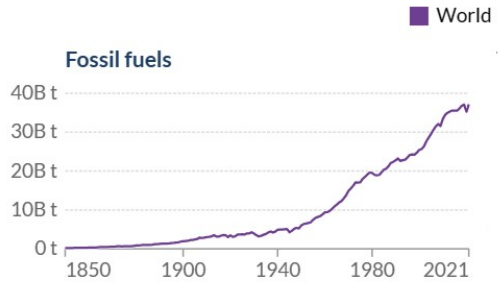


Figure 1.3: Global $CO_2$ emissions from fossil fuels.[6]

[5] https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/
[6] https://ourworldindata.org/grapher/global-co2-fossil-plus-land-use?facet=metric

The increase in $CO_2$ emissions leads to global warming and changes in climate and weather patterns, resulting in more floods, droughts, or intense rain, as well as more frequent and severe heat waves. The planet's oceans and glaciers are also affected—oceans are warming and becoming more acidic, ice caps are melting, and sea level is rising. These changes present challenges to our society and our environment, including health risks from heat waves, worsening air and water quality, and the spread of certain diseases.

Software, gaming services on various devices, servers, networks infrastructure and data centers generate a significant amount of carbon dioxide emissions. The servers and data centers also need to use a huge amount of energy to maintain the temperature so that the machines can work more efficiently. According to a study by Abraham, every company, studio, and developer he gathered data on including Ubisoft, Nintendo, and Microsoft were all somewhere in the range of generating 1 to 5 tons of $CO_2$ per year[7].

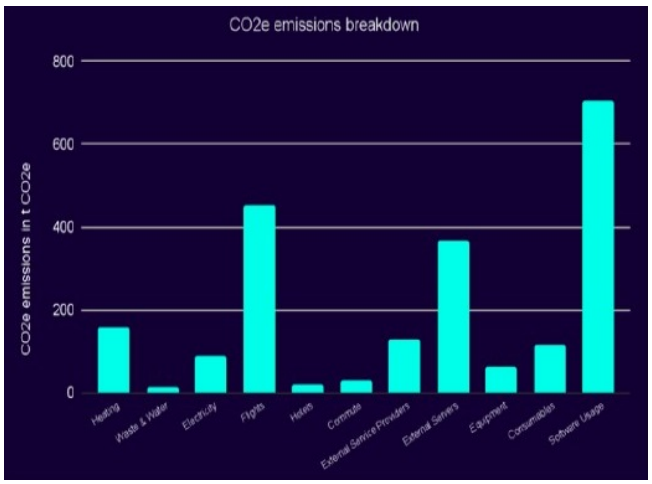


Figure 1.4: $CO_2$e emissions breakdown.[8]

To comprehend the energy consumption associated with software, the following examples are provided: In 2019, researchers found that the energy used to keep the Bitcoin network running was more than the energy used by the whole country of Switzerland[9]. Training a single neural network model today can emit as much carbon as five cars in their lifetimes[9]. Researchers trained an AI model to recognize different types of iris flowers. The model was 96.17% accurate and used 964 joules of energy. To make the model 1.74% more accurate, it needed 2,815 joules of energy. To make it just 0.08% more accurate, it needed almost 4 times more energy than the first stage[9].

Energy consumption plays a pivotal role in contributing to global carbon dioxide emissions, making it crucial to address this issue in the context of software development. While modifying user behavior to reduce energy consumption can be challenging, there are opportunities to assist developers and other stakeholders in integrating energy efficiency considerations throughout the software development life cycle.

In our research, we will undertake a thorough exploration of various tactics aimed at improving software energy efficiency. We will not only identify these tactics but also provide detailed insights

[7] https://www.polygon.com/features/22914488/video-games-climate-change-carbon-footprint
[8] https://www.planetly.com/articles/what-tech-companies-can-do-to-reduce-and-avoid-emissions
[9] https://hbr.org/2020/09/how-green-is-your-software

into their implementation, highlighting practical ways to integrate them into the software development process.

## 1.3 Problem Statement

The energy efficiency of software development is often overlooked by software developers, leading to suboptimal energy consumption in software applications. This lack of awareness and consideration for energy efficiency can be attributed to several factors:

- Currently, there is a lack of direct energy awareness that enable software developers to measure and understand the energy consumption of their source code during the development phase. This absence hinders the ability to identify and address energy inefficiencies early on.
- Software developers generally lack sufficient knowledge and awareness about how to save energy by optimizing their source code. Without the necessary guidance and information, they may unknowingly contribute to excessive energy consumption in their software applications.

Due to its platform independence and strong security features, Java dominated the programming language landscape as the most popular choice from 2015 to 2020. Despite its decline in recent years, Java still holds a significant position as the fifth most popular programming language and continues to be extensively utilized in server environments[10]. Though python is the most energy consuming Programming Language, it is also found by the researchers that Java also consumes more energy than C, C++. Given Java programming language's historical prominence, machine independence, security capabilities, and widespread adoption in server applications, we have selected Java as our primary focus for further research and development efforts.

## 1.4 Objectives

The main objective of this study is to explore tactics for enhancing software energy efficiency. From this objective we define 2 research questions:

- RQ1: Which tactics help to improve Energy Efficiency?
- RQ2: How can we automatise the integration of tactics to reduce energy consumption?
  - RQ2.1 Does the improvement of *execution time* and *memory consumption* reduce energy consumption?
  - RQ2.2 Could code refactoring integrate into GI? Which elements need to be extended in the Gin tool?
  - RQ2.3: In which extent code refactoring genetically improve the software to reduce energy consumption?

In response to Research Question 1 (RQ1), we explore tactics for improving energy efficiency, primarily focusing on code refactoring. We explore code refactoring as our main tactic due to its significant impact on energy conservation. We will delve into various code refactoring methods, including state-of-the-art techniques. Our ultimate goal is to elucidate how this tactical approach contributes to enhancing the energy efficiency of software.

[10] https://insights.stackoverflow.com/survey/2020#most-popular-technologies

In RQ2, we focus on automating the integration of tactics to lower energy consumption. The approach involves using a genetic algorithm, and as part of the research, a tool called GIN will be introduced. GIN is designed to improve existing software using search-based techniques. Three sub-questions have been identified: RQ2.1 aims to verify if improvements in response time and memory consumption(individually or together) can result in energy reduction. Experiments will be conducted using the GIN tool, with optimized versions compared to their original version using JoulerJX to assess energy impact. RQ2.2 investigates the feasibility of integrating code refactoring into Genetic Improvement (GI), identifying necessary extensions to the GIN tool. Finally, RQ2.3 explores the extent to which the integration of code refactoring can genetically enhance software to reduce energy consumption, testing its impact on energy efficiency.

## 1.5 Conclusion

The efficient utilization of smart devices and the transition towards sustainable energy sources have become crucial topics of global significance. In light of the challenge associated with altering user behavior and replacing existing devices, the emphasis lies in optimizing usage patterns. Numerous countries worldwide are actively working towards generating a larger proportion of their electricity from renewable sources, recognizing the potential to mitigate carbon emissions and combat global warming. However, this transition requires time and collective efforts. In the meantime, it is essential for individuals and communities to remain cognizant of their electricity consumption. By raising awareness and promoting energy efficiency, we can collectively contribute to reducing the strain on non-renewable resources and pave the way for a greener, more sustainable future.

This research comprises several distinct chapters. In the first chapter, the motivation behind the study, the objectives to be achieved, and the problem statement are discussed. This section establishes the context and rationale for the research, highlighting its relevance and significance within the field of study. Moving on to the second chapter, the background of the research is meticulously described. This chapter delves into the existing body of knowledge, theories, and prior studies relevant to the research topic, providing a comprehensive overview of the subject matter.

Chapter three provides a comprehensive assessment of research papers related to code refactoring, genetic improvement, and the Gin tool. It highlights the strengths and limitations of existing studies, justifies the choice of the Gin tool for program improvement, and identifies areas for extension to incorporate selected code refactoring techniques.

The chapter four is dedicated to the experimental work and the subsequent analysis of results. Here, the research methodology employed is presented, including the design of experiments, data collection procedures, and analysis techniques. The obtained results are thoroughly examined, with detailed explanations of the findings and their implications. This chapter aims to provide a comprehensive understanding of the experimental process.

The chapter five serves as the conclusion of the research. It consolidates the key findings, summarizing the main contributions of the study, and answering the research questions or hypotheses. Additionally, any limitations or challenges encountered during the research are acknowledged and discussed. Finally, recommendations for future work are presented in the chapter six. This section identifies potential areas for further investigation, suggesting avenues for future research that can build upon the current study and address any remaining gaps or unresolved questions.

# Chapter 2

# Background

## 2.1 Introduction

The initial motivation and primary focus of this research work is to explore tactics for enhancing software energy efficiency. With this in mind, we begin by trying to answer the following research question:

- **RQ1**: Which tactics help to improve Energy Efficiency?

In the subsequent paragraph, we will explain the answer to the posed research question.

Enhancing software energy efficiency is an important goal in the development of modern software systems. Several tactics can be used to achieve this goal, including Architectural Tactics, Code Optimization, Resource Adaptation, Design Patterns, and Code Refactoring.

- **Architectural Tactics**:These tactics focus on optimizing the software architecture for energy efficiency. For example, a study [Vos et al., 2022] by IEEE collaborated with a large cloud solution provider to discover an initial set of reusable architectural tactics for software energy efficiency. Starting from interviews with 17 practitioners, they reviewed and selected available tactics to improve the energy efficiency of individual workloads in the public cloud, and synthesized the identified tactics in a reusable model.

- **Code Optimization**:This tactic focuses on optimizing the code to improve energy efficiency. Techniques include computational efficiency, low-level or intermediate code optimization, parallelism, and data and communications efficiency. According to the Roskilde University's Energy-Aware Software Development Methods and Tools research paper, there are four categories of techniques for application software energy efficiency: Computational Efficiency, Low-level or Intermediate Code Optimization, Parallelism, and Data and Communications Efficiency[1].

- **Resource Adaptation**:This tactic focuses on optimizing hardware and software resource architectures to improve software efficiency. Strategies include reducing overhead, adapting services, and maximizing application efficiency. Software energy-efficiency resource adaptation strategies focus on optimizing hardware and software resource architectures to improve software efficiency[2].

- **Design Patterns**:These are reusable solutions to common problems in software design that can be used to improve energy efficiency. For example, a study [Schaarschmidt et al., 2020] by

[1] https://medium.com/@maxmeinhardt/software-energy-efficiency-code-optimization-tactics-b95be4ffcaf7
[2] https://medium.com/@maxmeinhardt/software-energy-efficiency-resource-adaptation-tactics-8a2063c96213

Springer presents an energy-aware software design pattern framework description, which takes power consumption and time behavior into account. They evaluate the expressiveness of the framework by defining design patterns, which use elaborated power-saving strategies for various hardware components to reduce the overall energy consumption of an embedded system.

- **Code Refactoring**:This is the process of restructuring existing code without changing its external behavior to improve its internal structure. Refactoring techniques aim to reduce the energy consumption of the software. A study [Şanlıalp et al., 2022] by MDPI examines the effect of code refactoring techniques on energy consumption. A total of 25 different source codes of applications programmed in the C# and Java languages are selected for the study, and combinations obtained from refactoring techniques are applied to these source codes. The results show that the combinations significantly improve the software's energy efficiency.

We are providing a comparative analysis between all the mentioned tactics:

| **Tactic** | **Purpose** | **Pros** | **Cons** | **Reference** |
|---|---|---|---|---|
| Architectural Tactics | Support cloud consumers in developing energy-efficient workloads in the public cloud by discovering an initial set of reusable architectural tactics for software energy efficiency. However, the process is not yet straightforward due to the current lack of transparency of cloud providers. | Can improve the overall architecture of the system and make it more efficient, reduced energy consumption, cost savings, environmental benefits. | Cloud consumers do not have full access to information regarding their cloud infrastructure usage, which is required to understand the impact of design decisions on energy usage, limited research available on how cloud consumers can reduce their energy footprint when running software in the public cloud and may require significant changes to the system's architecture. | [Vos et al., 2022] |
| Code Optimization | Improve the performance of the code by making it consume fewer resources such as CPU, memory, and disk space. This can result in faster execution and improved energy efficiency. | Improved performance, reduction in code size, cleaner code base. | Code optimization can reduce readability, it can be a time-consuming process and may delay the overall compiling process. | [University et al., 2016] |
| Resource Adaptation | Optimize hardware and software resource architectures to improve software efficiency, reduce overhead, adapt services, and maximize application efficiency. | Reduced overhead and improved efficiency, improved energy efficiency, reduced costs. | Require significant effort and resources to implement and may not be suitable for all types of software or systems. | [Vos et al., 2022] |
| Design Patterns | Optimize the software design in early development phases, taking into account energy consumption and time behavior in the context of energy efficiency. | Reduce the overall energy consumption of an embedded system, improve energy efficiency, and potentially extend the equipment life. | One potential con is that the use of design patterns may not always result in improved energy consumption. For example, a study found that for the majority of cases, alternative designs excelled in terms of energy consumption compared to pattern solutions. | [Schaarschmidt et al., 2020] and [Feitosa et al., 2017] |

| | | | | |
|---|---|---|---|---|
| Code Refactoring | Restructure existing code without changing its external behavior to enhance reusability and maintainability of software components through improving nonfunctional attributes of the software. | Improve code readability, reduce complexity, and make the code more efficient, maintainable, and easier to understand. Some refactoring techniques aim to reduce the energy consumption of the software, which can improve its energy efficiency.Refactoring transforms a mess into clean and simple code. | Can introduce new bugs or fail to preserve the external functionality of the code, If code refactoring is not done carefully. | [Şanlıalp et al., 2022] and [Kim et al., 2018a] |

Table 2.1: Comparison between all mentioned tactics

After comparing all tactics, we choose code refactoring because it offers several advantages over others. Unlike other tactics, code refactoring does not require significant architectural changes, has no major resource limitations, and maintains readability. Additionally, it suits various types of software. By applying code refactoring, we can improve code simplicity, readability, reduce complexity, and enhance code efficiency, leading to improved energy efficiency. This answers our **RQ1**. Further exploration of code refactoring for energy-efficient software will be discussed in chapter 3.

Monitoring energy use is a crucial prerequisite before embarking on our main experiments, as accurately measuring energy consumption is paramount. Without understanding the energy usage of programs, progress to our main experiments becomes uncertain. It is essential to determine which tool will be the most efficient for measuring energy consumption. To achieve this, we explore various tools, each with its advantages and drawbacks. In this subsequent section, we will discuss the advantages and drawbacks of various tools.

## 2.2 Energy Consumption Profiling Tools

Measuring energy consumption directly is challenging because there's no straightforward way for directly measuring energy usage. To overcome this, we rely on external tools to measure energy. Energy consumption measuring tools can be categorized mainly in two categories:

- **Hardware Tools**: Wattmeter [Bekaroo et al., 2014]
- **Software Tools**:
  - Power Joular [Noureddine, 2022]
  - JoularJX [Noureddine, 2022]
  - Likwid Powermeter[3] [Treibig et al., 2010]

As our aim is to determine which tool will be the most efficient for measuring the energy consumption of a software program, we will mainly focus on software tools. Now, we will provide a comparative analysis of all the aforementioned software tools.

[3] https://github.com/RRZE-HPC/likwid/wiki/Likwid-Powermeter

| Software Tools | Purpose | Advantages | Disadvantages |
|---|---|---|---|
| Power Joular [Noureddine, 2022] | PowerJoular monitors the power consumption of CPU and GPU for PCs, servers, and single-board computers (such as Raspberry Pi) | Monitor the power consumption of CPU and GPU for PCs, servers, and single-board computers. It can monitor the power consumption of individual processes in GNU/Linux and expose power consumption to the terminal and CSV files. It provides a systemd service (daemon) to continuously monitor the power of devices with low overhead. | Currently only works on GNU/Linux platforms. Additionally, it requires root/sudo access on the latest Linux kernels (5.10 and newer) |
| JoularJX [Noureddine, 2022] | JoularJX is a Java-based agent for software power monitoring at the source code level. It provides real-time power consumption of every method in the monitored program and total energy for every method on program exit. | Monitor power consumption of each method at runtime, uses a Java agent, no source code instrumentation needed, uses Intel RAPL (powercap interface) for getting accurate power reading on GNU/Linux, research-based regression models on Raspberry Pi devices, and a custom program monitor (based on Intel Power Gadget) for accurate power readings on Windows, provides real-time power consumption of every method in the monitored program, provides total energy for every method on program exit. | JoularJX requires a minimum version of Java 11+. Additionally, JoularJX depends on specific software or packages in order to get power or energy readings. JoularJX can only measure the energy and power consumption for the Java source codes and applications. |
| Likwid Powermeter [Treibig et al., 2010] | Likwid Powermeter is a tool for accessing RAPL (Running Average Power Limit) counters on Intel processors, which allows you to query the energy consumed within a package for a given time period and computes the resulting power consumption. | Monitor the energy consumption by the core of the CPU of the machine, provide the result by measuring the energy consumption by each processor. | Not able to monitor the energy consumption method-wise. Provide processor-wise results (at the level of CPU cores), sometimes results can vary as other processes may run on the same core with the test process. |

Table 2.2: Comparison between all mentioned energy consumption monitoring software tools.

After comparing all the mentioned energy consumption monitoring software tools, we chose JoularJX to run our experiments and measure energy and power consumption. The key reason for selecting JoularJX is that it allows real-time monitoring at the source code level without requiring source code instrumentation. It functions as a Java agent, providing accurate power and energy readings on both GNU/Linux and Windows platforms. This makes it a suitable tool for monitoring the energy consumption of Java-based software or Java-based programs. As mentioned in Chapter 1, Section 1.3, our primary focus is on Java programs or Java-based software for making energy-efficient, which makes JoularJX the appropriate choice for our requirements.

## 2.3 Studies of refactoring for energy efficiency

In preliminary work, we conducted energy consumption measurements on two Java programs using JoularJX as our energy consumption monitoring tool. We have chosen Ray-casting Algorithm mentioned by the authors in the research paper [Noureddine, 2022] and Mandelbrot set Algorithm. The programs are presented:

- **Ray-casting Algorithm**[4]:Ray-Casting algorithm is widely used in 2D computer graphics and collision detection. The algorithm efficiently determines whether a point is located inside or outside a given polygon by examining the intersections of rays originating from the point and the polygon's edges.

[4] https://rosettacode.org/wiki/Ray-casting

- **Mandelbrot set Algorithm**[5]:The Mandelbrot set is a two-dimensional set with a relatively simple definition that exhibits great complexity, especially as it is magnified. It is popular for its aesthetic appeal and fractal structures.

| **Algorithm Name** | **Consumed energy (Joules)** |
|---|---|
| Ray-casting Algorithm | 10.8124 Joules [6] |
| Mandelbrot set Algorithm | 126.2217 Joules [7] |

Table 2.3: Energy consumption of different algorithms

We also tested a Java project called cmath using JoularJX. We will explain more about our experiment on monitoring the energy consumption of a Java program (Mandelbrot set Algorithm) and a Java project (cmath project) in Chapter 4, section 4.1.

## 2.4 Conclusion

We were able to successfully measure the energy consumption of individual methods in the Java programs we tested. Based on our findings, we can make some conclusions.

- We prepared a shell script to measure the energy consumption of Java programs, which was able to run each program 30 times with a parameter value of 15000. After that, the output data was stored in the 'energy_filtered' and 'power_filtered' directories under the 'jx_results' folder. These directories contain processed data related to the energy and power consumption of the programs, monitored using JoularJX.
- In the Ray-casting Algorithm experiment, a method of the JVM (Java Virtual Machine) named java.lang.ref.Reference.waitForReferencePendingList was found to consume the maximum amount of energy. This method was also present in the Mandelbrot set Algorithm experiment and consumed a significant level of energy.
- In the Mandelbrot set Algorithm experiment, the getByte() method was consuming the maximum amount of energy.

[5] https://en.wikipedia.org/wiki/Mandelbrot_set
[6] https://drive.google.com/drive/folders/1XKNE7mrzMzVqS-V0tyaJ7QHJZXQBrCQx
[7] https://drive.google.com/drive/folders/1OwDY9FucaXNVOdpCPoxTV-3WB698i4u2

# Chapter 3

# Literature Review: Code Refactoring, Genetic Improvement and Gin Tool

## 3.1 Introduction

Conducting a literature review is an essential step in any research project. In this section, we carefully reviewed a number of research papers on related topics, including code refactoring for software energy efficiency and genetic improvement tool for obtaining the best patch to improve program performance. Through this review, we assessed the strengths and limitations of the existing research. The insights gained from this review were valuable in identifying potential code refactoring techniques for integration into the gin tool. Additionally, we were able to justify our selection of the gin tool for obtaining an improved version of the program. Furthermore, this literature review assisted us in identifying specific components of the gin tool that need extension to accommodate the selected code refactoring techniques, which aligns with Research Question 2.2.

## 3.2 Code Refactoring and Energy Efficiency

In this section, we will review the existing research on code refactoring, exploring its significance and advantages. Additionally, we will investigate various available code refactoring techniques, conducting a comparative analysis to identify the most suitable code refactoring techniques. Subsequently, we will select the suitable code refactoring techniques for integration into the gin tool, based on the findings of our comparison.

### 3.2.1 Code Refactoring

Code refactoring is the process of restructuring existing computer code without changing its external behavior to enhance reusability and maintainability of software components through improving nonfunctional attributes of the software [Kim et al., 2018a]. It is a way to improve the code quality and maintainability[1]. The benefits of code refactoring include removing bad smells, reducing code size, improving readability, and making it easier to enhance and maintain in the future. However, there are also limitations to code refactoring. For example, it can be expensive and risky in the view of management, may introduce bugs, and can be difficult to do if the code is already a big mess.

[1] https://www.c-sharpcorner.com/article/pros-and-cons-of-code-refactoring/

Additionally, refactoring should not be done if a deadline is near or if the cost of refactoring is higher than rewriting the code from scratch[1].

In the context of energy efficiency, code refactoring can be used to improve the energy consumption of software by making changes to the source code that reduce its energy usage. Several studies have investigated the impact of code refactoring on energy consumption. For example, a study [Sahin et al., 2014] presents an empirical study to investigate the energy impacts of 197 applications using 6 commonly-used code refactoring methods. The results show that code refactoring methods can not only impact energy usage but can also increase and decrease the amount of energy used by an application. Another study [Ournani et al., 2021] found that developers can optimize the power consumption of software by improving their source code implementations. Furthermore, a study [Morales et al., 2018] proposed a novel anti-pattern correction approach called EARMO. In this study, the researchers analyzed the impact of eight types of anti-patterns on a testbed of 20 android apps extracted from F-Droid and evaluated EARMO using three multiobjective search-based algorithms. The results showed that EARMO can generate refactoring recommendations in less than a minute and remove a median of 84 percent of anti-patterns. Moreover, EARMO extended the battery life of a mobile phone by up to 29 minutes when running a refactored multimedia app with default settings (no Wi-Fi, no location services, and minimum screen brightness) in isolation. A qualitative study was also conducted with developers of the studied apps to assess the refactoring recommendations made by EARMO. Developers found 68 percent of refactorings suggested by EARMO to be very relevant. Additionally, code smells, which are symptoms of poor design or implementation choices, have been found to impact energy consumption. A study [Palomba et al., 2019] found that refactoring certain code smells reduced energy consumption by up to 10.8%.

According to the information mentioned above, it can be concluded that code refactoring can have a positive impact on energy efficiency by reducing the energy consumption of software. However, the benefits and limitations of code refactoring for energy efficiency may vary depending on the specific context and implementation.

### 3.2.2 Types of Code Refactoring Techniques and Comparison between mentioned Code Refactoring Techniques

In Table 3.1 below, we present the different types of code refactoring Techniques and compare them based on energy consumption impact

| **Code Refactoring Techniques Name** | **Purpose** | **Energy Consumption Impact** |
|---|---|---|
| Convert Local Variable to Field [Sahin et al., 2014] | Turns a local variable into a field. | This refactoring had a statistically significant difference in energy usage in some cases for JVM. This means that in some cases, applying this refactoring resulted in a noticeable change in the amount of energy used by the program. In most cases, applying this refactoring resulted in an decrease in the amount of energy used by the program. |
| Extract Local Variable [Sahin et al., 2014] | Creates a new variable assigned to the selected expression and replaces the selection with a reference to the new variable. | It had the least impact on energy usage, with only a few cases showing a significant difference. This means that in most cases, applying this refactoring did not result in a noticeable change in the amount of energy used by the program. In some cases, for JVM 6, the amount of energy used by the program decreases |
| Extract Method [Sahin et al., 2014] | Creates a new method containing the selected statement or expression and replaces the selection with a reference to the new method. | This refactoring increased energy usage 8 times and decreased energy usage 2 times on JVM6. This means that in most cases, applying this refactoring resulted in an increase in the amount of energy used by the program. However, in a few cases, there was a decrease in energy usage after applying this refactoring. |

| | | |
|---|---|---|
| Inline Method [Sahin et al., 2014] | Copies the body of a callee method into the body of a caller method. | It had varying impacts on energy usage across different applications and platforms. This means that the impact of this refactoring on energy usage is not consistent and can vary depending on the specific application and platform. In some cases, it may reduce energy usage, while in others it may increase it. |
| Introduce Indirection [Sahin et al., 2014] | Creates a static method to indirectly delegate to the selected method. | This refactoring had both positive and negative impacts on energy usage. This means that the impact of this refactoring on energy usage is not consistent and can vary depending on the specific case. In most cases, applying this refactoring resulted in an increase in the amount of energy used by the program. However, in a few cases, there was a decrease in energy usage after applying this refactoring. |
| Introduce Parameter Object [Sahin et al., 2014] | Replaces a set of parameters with a new class and updates all callers to pass an instance of the new class as the value to the introduced parameter. | It had a significant impact on energy usage. This means that in most cases, applying this refactoring resulted in a decrease in the amount of energy used by the program. |
| Inline Temp [Barack and Huang, 2018] | Replace all references to a temporary variable with the expression that was assigned to it. This can improve code readability and reduce the number of variables in the code. | Eliminating temporary variables speeds up the fetching of redundant temporary variables from both the main and cache memories. This approach improves performance and maintains the same level of energy efficiency, which is considered a positive improvement. |
| Move Method [Morales et al., 2018] | Improve the organization of code by moving a method to a more appropriate class or object. | Can reduce energy consumption by improving the efficiency of the code. |
| Simplify Nested Loop [Kim et al., 2018b] | Reduce the dimensions of multi-dimensional loops. | Dimension reduction of multi-dimensional loops helps to reduce energy consumption and make the code more readable and energy-efficient. |
| Encapsulate Field [Park et al., 2014] | Set access permissions of a variable by creating getters and setters for the selected field, allowing the field to be accessed and modified only through these methods. This provides better control over the field and can improve the maintainability of the code. | This technique can have an impact on energy efficiency, especially when combined with other code refactoring techniques. These combinations can provide significant improvements in energy efficiency. |

Table 3.1: Types of Code Refactoring Techniques

### 3.2.3 Analysis and Selection of Code Refactoring Methods for Integration into the Gin Tool

The analysis of various code refactoring techniques, as presented in the table 3.1, reveals distinct patterns in their impact on energy consumption. Several techniques, including Convert Local Variable to Field, Introduce Parameter Object, Inline Temp, Move Method, Simplify Nested Loop and Encapsulate Field, consistently show positive results in reducing energy consumption across different scenarios. Notably, the combined application of Inline Temp, Simplify Nested Loop, and Encapsulate Field techniques proves to be particularly effective in enhancing energy efficiency [Şanlıalp et al., 2022].

Conversely, Extract Local Variable, Extract Method, Inline Method, and Introduce Indirection exhibit varying effects on energy consumption depending on the specific context. Hence, it is essential to apply these techniques with caution and consider the unique circumstances of each codebase.

For the integration of code refactoring techniques into the Gin Tool to optimize energy efficiency, our top priority would be to incorporate Convert Local Variable to Field and Introduce Parameter Object code refactoring techniques, which consistently yield energy reductions for Java programming. Additionally, the Move Method code refactoring technique, primarily used for Mobile Apps, can be considered for Java applications to enhance energy efficiency further. Finally, the integration of

Inline Temp, Simplify Nested Loop, and Encapsulate Field code refactoring techniques would be valuable, as they have demonstrated consistent energy consumption improvements in various cases. By making these selective and informed choices, we aim to enhance the Gin Tool's capabilities and provide developers with effective means to optimize energy consumption in their codebases.

### 3.2.4 Conclusion

We have explored a range of code refactoring techniques and their implications for energy efficiency in software systems. While all methods show potential in improving certain aspects of software performance, not all contribute equally to energy efficiency. Techniques such as 'Convert Local Variable to Field', 'Introduce Parameter Object', 'Inline Temp', 'Move Method', 'Simplify Nested Loop', and 'Encapsulate Field' have demonstrated significant potential in reducing energy consumption. However, the actual impact on energy efficiency may vary based on the context, which includes factors like the nature of the codebase, the specific implementation of refactoring, and the overall software architecture. In the subsequent section, we will identify specific components of the gin tool that need to be extended to integrate the selected code refactoring techniques, with the aim of improving energy efficiency without compromising other crucial software characteristics.

## 3.3 Genetic Improvement (GI) and Gin Tool

In this section, we will discuss the Genetic Improvement (GI) and Gin Tool. We will explore the strengths and limitations of existing research on genetic improvement and gin tool. We will also justify our selection of the gin tool for obtaining an improved version of the program based on the best patch for lower fitness function and identify specific components of the gin tool that need extension to integrate the selected code refactoring techniques, which aligns with Research Question **RQ2.2**.

### 3.3.1 Genetic Improvement (GI)

Genetic Improvement (GI) is a field of research that uses automated search to find improved versions of existing software [Brownlee et al., 2019] [Petke et al., 2018]. It can improve both functional properties of software, such as bug repair, and non-functional properties, such as execution time, energy consumption, or source code size [Zuo et al., 2022]. Researchers have already shown that GI can improve human-written code, ranging from program repair to optimizing run-time, from reducing energy consumption to the transplantation of new functionality [Brownlee et al., 2019].

Numerous research studies have been conducted in the Genetic improvement domain. In Table 3.2 below, we present a compilation of relevant research works that are well-suited for supporting and enhancing our own research exploration. The table includes key information about each work, such as title, author name, and key findings.

| Title | Author Name | Key Findings |
|---|---|---|
| Genetic Improvement of Software: A Comprehensive Survey [Petke et al., 2018] | Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. | Present a comprehensive survey of the nascent field of research on Genetic Improvement (GI) with a focus on the core papers in the area published between 1995 and 2015. Authors identified core publications including empirical studies, 96% of which use evolutionary algorithms (genetic programming in particular). GI has resulted in performance improvements for a diverse set of properties such as execution time, energy and memory consumption, as well as results for fixing and extending existing system functionality. |

| | | |
|---|---|---|
| Gin: Genetic Improvement Research Made Easy [Brownlee et al., 2019] | Alexander E.I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. | Introduced an extensible and modifiable GIN toolbox for GI experimentation, with a novel combination of features. Instantiated in Java and targeting the Java ecosystem, Gin automatically transforms, builds, and tests Java projects and supports automated test generation and source code profiling. The study showed through examples and a case study how Gin facilitates experimentation and can speed up innovation in GI. |
| Evaluation of Genetic Improvement Tools for Improvement of Non-functional Properties of Software [Zuo et al., 2022] | Shengjie Zuo, Aymeric Blot, Justyna Petke | The study conducted a literature review of available GI tools and ran multiple experiments on the found open-source tools to examine their usability. It applied a cross-testing strategy to check whether the available tools can work on different programs. Overall, the study found 63 GI papers that use a GI tool to improve non-functional properties of software, out of which 31 are accompanied by open-source code. The study was able to successfully run eight GI tools and found that ultimately only two, Gin, and PyGGI, can be readily applied to new general software. |
| Multi-Objective Genetic Improvement: A Case Study with EvoSuite [Callan and Petke, 2022] | James Callan and Justyna Petke | Present an extension of an existing generalist, open-source genetic improvement tool, Gin, with a multi-objective search strategy, NSGA-II. The implementation was conducted on a mature, large software, EvoSuite, a tool for automatic test case generation for Java. The multi-objective extension of Gin was utilized to improve both the execution time and memory usage of EvoSuite. The study found improvements in the execution time of up to 77.8% and improvements in memory usage of up to 9.2% on the mentioned test set. |
| Reducing Energy Consumption Using Genetic Improvement [Bruce et al., 2015] | Bobby R. Bruce, Justyna Petke and Mark Harman | Applied GI to the MiniSAT Boolean satisfiability solver when specializing for three downstream applications and found that GI can successfully be used to reduce energy consumption by up to 25%. |
| Evolutionary Approximation of Software for Embedded Systems: Median Function [Mrazek et al., 2015] | Vojtech Mrazek, Zdenek Vasicek, Lukas Sekanina | The study presented a method based on Cartesian genetic programming that is evaluated in the task of approximation of 9-input and 25-input median function. Resulting approximations shown a significant improvement in the execution time and power consumption with respect to the accurate median function while the observed errors are moderate. |
| Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guav [Burles et al., 2015] | Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan and Nadarajen Veerapen | In this study, the authors used a metaheuristic search to improve Google's Guava library by finding a semantically equivalent version of com.google.common.collect.ImmutableMultimap with reduced energy consumption. Semantics-preserving transformations were found in the source code using the principle of subtype polymorphism. A new tool, Opacitor, was introduced to deterministically measure energy consumption and it was found that a statistically significant reduction to Guava's energy consumption is possible. |

Table 3.2: Genetic Improvement Research Papers Summary

After a thorough analysis of existing research work on Genetic Improvement (GI), it becomes evident that Genetic Improvement (GI) is a powerful technique that can be used to automatically find improved versions of existing software with respect to various non-functional properties such as execution time, energy consumption, memory usage, etc. Several tools have been developed to facilitate experimentation with GI, including Gin and PyGGI. These tools have been successfully applied to various software systems, resulting in significant improvements in performance. In the subsequent subsection 3.3.2, we will discuss about the selection of Gin tool and justify why this tool will be suitable for our work.

### 3.3.2 Justify the selection of the Gin Tool for the improvement of non-functional properties of software

According to the research work [Zuo et al., 2022], the authors found 63 GI papers that used a GI tool to improve non-functional properties of software, of which 31 had associated open-source code. The

usability study of this research exposed 11 different general GI tools, but only 8 were able to run without any issues. Furthermore, the generalizability study of this research ultimately showed that among these eight GI tools, only two, Gin and PyGGI, can be readily applied to new software for the improvement of non-functional properties. The Gin tool[2] provides an extensible and modifiable toolbox for GI experimentation, specifically targeting the Java ecosystem. On the other hand, the PyGGI tool[3] offers a Python General lightweight and simple framework for Genetic Improvement. In the subsequent section we will discuss about Gin tool.

As mentioned in Chapter 1 and Section 1.3, we have selected the Java programming language as our primary focus for our research work. Therefore, we preferred a Java-based Genetic Improvement tool instead of a Python-based Genetic Improvement tool. That's why we selected the Gin tool, a Java-based Genetic Improvement tool, to help us improve the non-functional properties of software. In the subsequent subsection 3.3.3 we will discuss about Gin Tool.

### 3.3.3 Gin Tool

Gin is a genetic improvement (GI) tool that aims to facilitate experimentation and research in the field of software development. It provides an extensible and modifiable toolbox for GI experimentation, specifically targeting the Java ecosystem. By automating the transformation, building, and testing of Java projects, Gin supports various aspects of software improvement, including program repair, runtime optimization, energy consumption reduction, and the addition of new functionality. The tool incorporates features such as automated test generation and source code profiling, which are essential for non-functional improvement. Gin's design focuses on scalability, allowing it to handle large-scale systems and integrate with popular Java build systems like Maven and Gradle. It supports multiple representations of code, providing flexibility for researchers to define custom mutation operators and transformation strategies. Additionally, Gin introduces innovative features for non-functional improvement, including built-in profiling and automated test case generation.

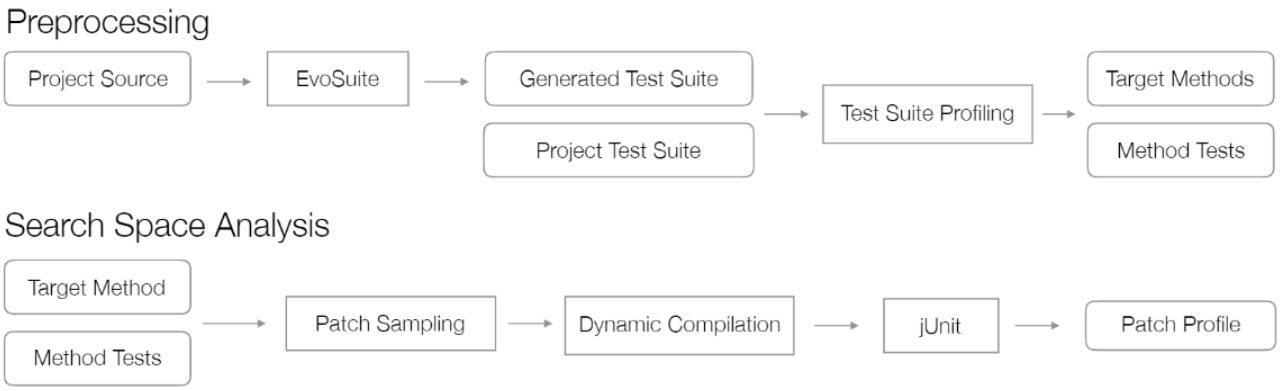


Figure 3.1: Gin Pipelines [Brownlee et al., 2019]

As shown in Figure 3.1, Gin provides two example pipelines for Preprocessing and Search Space Analysis.

**Preprocessing**: Gin can preprocess a project and find the methods that are most likely to benefit

[2] https://github.com/gintool/gin
[3] https://github.com/coinse/pyggi

from genetic improvement (GI). This is done by using the gin.util.Profiler class, which measures the execution time of each method in the project and ranks them by their contribution to the overall performance. The methods with the highest execution time are called 'hot methods' and are output as suitable targets for improvement by GI.

**Search Space Analysis**: Gin can also help to analyze the search space of possible program edits that can be applied by GI. The toolkit provides several tools that can sample and enumerate different types of edits, such as statement deletion, insertion, or replacement. These tools can be easily extended or reused to add new edit types. Gin will test each sampled or enumerated edit by applying it to the original code and running a test suite against the modified code. Gin will record various information about each edit, such as its validity (whether it preserves the functionality of the original code), its compilation result, its test output (whether it passes or fails the test suite), its run time (how long it takes to execute the test suite), and its error details (if any). Gin can use any test suite that is in JUnit format, which is a widely used testing framework for Java. Gin can also capture more detailed test output than just pass or fail, such as the difference between the expected and actual output. This allows Gin to support more fine-grained fitness functions that can measure the quality of each edit more accurately.

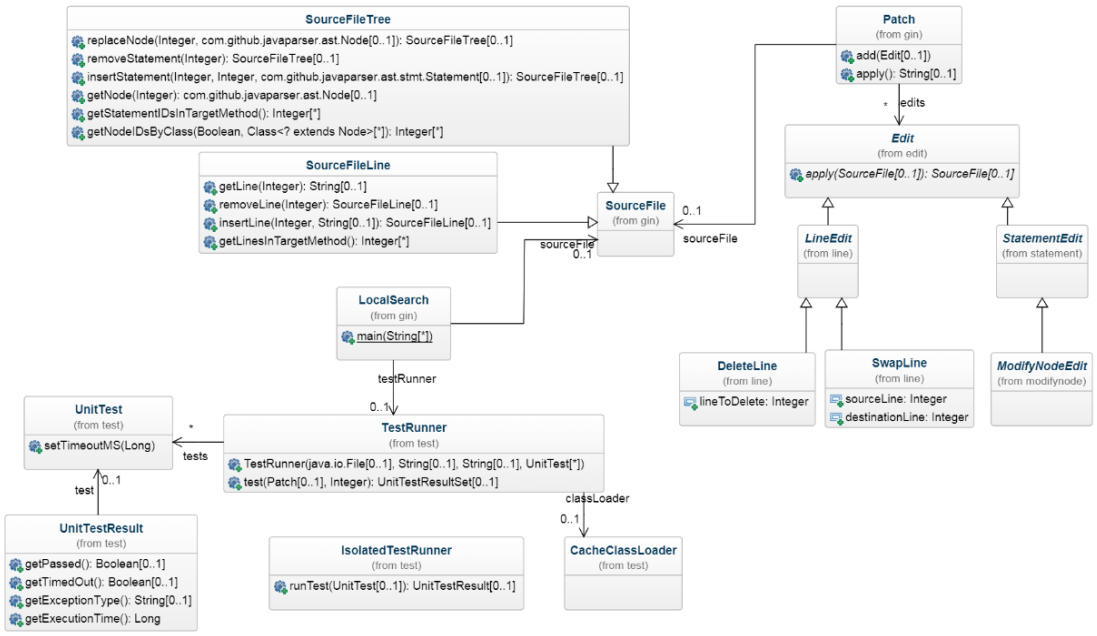


Figure 3.2: Gin Core Classes. [Brownlee et al., 2019]

The Gin toolkit encompasses a collection of classes mentioned in Figure 3.2 designed to facilitate genetic improvement research by offering a framework for manipulating source code, executing tests, and analyzing outcomes. At the core of this toolkit lies the SourceFile class, an immutable representation of the original source code, equipped with various methods for modifying the codebase, accessing language constructs, and generating modified Java source code.

Derived from the SourceFile class, the SourceFileTree subclass focuses on edits to the Abstract Syntax Tree (AST) of the source code. It assigns unique identifiers to each node within the AST, enabling efficient resolution of patches that entail multiple edits to the same location. In a similar vein, the SourceFileLine subclass directs its attention to line-level edits, also employing unique IDs

for each line to simplify edit application.

A crucial component within Gin is the Patch class, which serves as a container for a series of edits, encapsulating the desired changes to be applied to the source code. The Edit class, serving as the base class for different types of edits, represents the application of a specific operator to the target source code. Subclasses of Edit, including LineEdit and StatementEdit, offer a range of operations for manipulating lines of code and modifying statements, respectively. The granularity of these edits provides fine-grained control over the code transformations.

To explore the search space of software transformations, the LocalSearch class employs a combination of sampling and searching techniques. It navigates through possible modifications to improve the code, ultimately enhancing its quality and performance. Meanwhile, the TestRunner class utilizes the JUnit framework to execute unit tests, offering insights into the outcomes, execution time, and encountered errors of the modified code.

The UnitTest class represents an individual unit test and is employed by the TestRunner to evaluate the test outcomes. Storing the result of a unit test, including pass/fail status, expected and actual results, and error details, the UnitTestResult class aids in analyzing the impact of code modifications on test behavior.

For focused testing of individual edits or patches, the IsolatedTestRunner subclass of TestRunner conducts tests in isolation. Finally, the CatcheClassLoader class, a custom ClassLoader, loads the modified class during test execution, overlaying the existing class hierarchy and facilitating the loading of the modified class by JUnit.

Collectively, these classes and their interplay within the Gin toolkit provide researchers with a powerful foundation for genetic improvement studies. The toolkit simplifies the process of editing source code, conducting tests, and evaluating the effects of modifications, thereby enabling more efficient and effective research in this domain.

### 3.3.4 Identify the element need to be extended in the Gin tool

In the previous subsection, subsection 3.3.3, we mentioned that the Gin tool comprises a collection of classes. Among these classes, the Edit class serves as the base class for different types of edits, representing the application of specific operators to the target source code. The subclasses of Edit, namely LineEdit and StatementEdit, offer a range of operations for manipulating lines of code and modifying statements, respectively. Based on the information mentioned above, it can be concluded that code refactoring techniques can be integrated into the genetic improvement tool, Gin Tool.

According to the information provided, the Edit class in the Gin tool acts as the base class for different types of edits and has two subclasses, LineEdit (providing a variety of operations for manipulating lines of code) and StatementEdit (offering a range of operations for modifying statements). Consequently, we have identified the StatementEdit class as the appropriate class for integrating selected code refactoring techniques. As StatementEdit offers a range of operations for modifying statements, and code refactoring techniques are closely related to statement edits, we consider StatementEdit to be the suitable class for incorporating code refactoring techniques. This integration will aid in obtaining the best patch, indicating an improved version of the program that consumes less energy.

Figure 3.3 below illustrates the identified class of the Gin tool where we will integrate the selected code refactoring techniques.

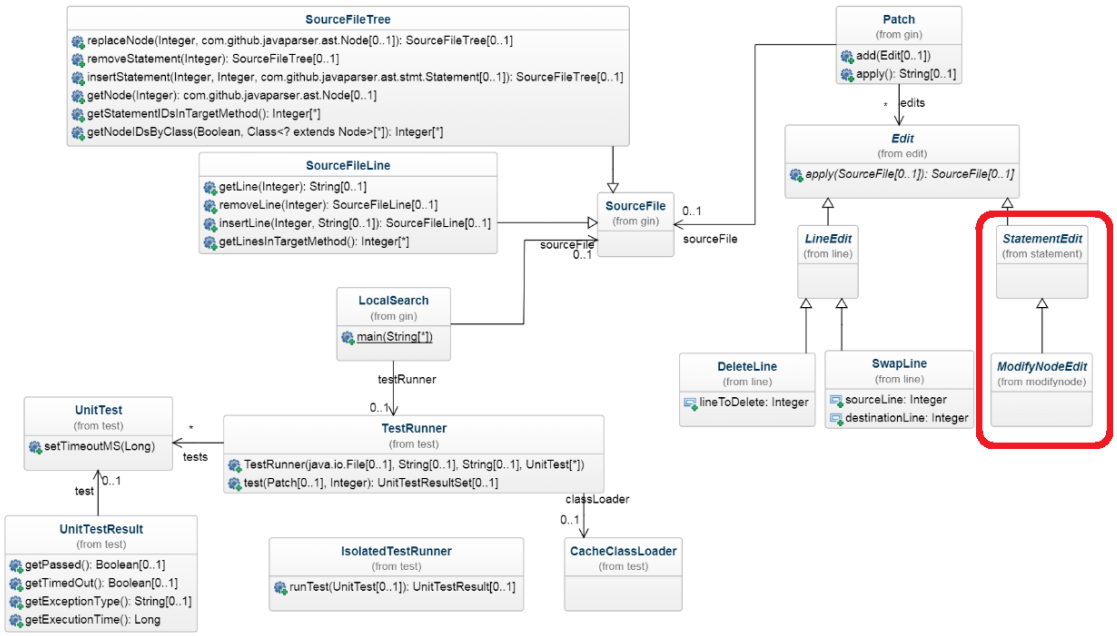


Figure 3.3: Extension of the Gin Tool's class. [Brownlee et al., 2019]

### 3.3.5 Conclusion

Code refactoring techniques and the Gin tool both aim to improve the quality of software. Code refactoring techniques improve the internal structure of the code without changing its external behavior, while Gin applies Genetic Programming and other Metaheuristics to existing software to improve it. Both can help improve the code's performance, maintainability, and scalability. It is possible that such integration(Code refactoring techniques integration in the Gin tool) could be beneficial in improving the quality of software. Integrating code refactoring techniques into Gin could potentially enhance its capabilities and provide additional ways to improve software quality. As Gin tool's StatementEdit class offers a range of operations for modifying statements, and code refactoring techniques are closely related to statement edits, we consider StatementEdit to be the suitable class for integrating code refactoring techniques.

Based on this analysis, we can answer research question **RQ2.2**: Code refactoring techniques can indeed be integrated into the Genetic Improvement tool, Gin. For this integration, the StatementEdit class in Gin will need to be extended.

## 3.4 Conclusion

This literature review examined research papers on code refactoring for energy efficiency and the use of the Genetic Improvement (GIN) tool to enhance program performance. Through the evaluation of the strengths and limitations of existing studies, we gained crucial insights into potential code refactoring techniques for the GIN tool. The review justified our choice of the GIN tool and facilitated the identification of the suitable class of GIN tool for extension, enabling the integration of selected refactoring techniques. This process successfully answered Research Question **RQ 2.2**.

# Chapter 4

# Genetic Improvement toward Energy Efficiency

## 4.1 Experimental executions using JoularJX

Our main objective is to reduce energy consumption in software. In order to delve deeper into this subject, it is essential to gain a comprehensive understanding of how we can accurately measure the energy consumption of a program or project. Determining the energy consumption is of utmost importance as it provides a baseline for our efforts to minimize it. To measure the energy consumption of a single Java program, we can hook JoularJX to the Java Virtual Machine when starting our Java program. For example: java -javaagent:joularjx-$version.jar selectedProgram. JoularJX will then monitor the power consumption of each method at runtime, providing real-time power consumption data for every method in the monitored program and total energy for every method on program exit. We can also use JoularJX to monitor the power consumption of each method in an entire project at runtime, providing detailed information about the energy consumption of the project as a whole. If we are using test suites to evaluate the energy consumption of the project, we can run the test suite with JoularJX hooked to the Java Virtual Machine to monitor the power consumption during testing.

As part of our research experiment, we conducted two preliminary experiments using JoularJX, a tool for monitoring energy consumption at the source code level:

- **Energy consumption monitoring of a single java program (mandelbrot.java) using JoularJX.**
- **Energy consumption monitoring of a project (cMath_original) using JoularJX.**

Now, we will explain detail about the experiments conducted as part of our research project.

### 4.1.1 Energy consumption monitoring of a single java program (mandelbrot.java) using JoularJX:

In this experiment, we monitored the energy consumption of a single Java program, mandelbrot.java using JoularJX. We used JoularJX 2.0 version to obtain updated tree structure results in the output directory. To execute the Java program with JoularJX, we prepared a bash script named "jx_script.sh", which includes several Python files: "jx_gatherData.py," "jx_plot.py," "jx_process_level_methods.py," "shapiro_wilk_test_energy.py," and "shapiro_wilk_test_power.py.". Now, we will explain how the script works to collect the data of energy consumption of the program (mandelbrot.java).

The script began by checking if a single argument, which was expected to be the name of the program without the file extension, was provided. If not, an error message was displayed. Then, a series of directories were created to organize the results. These directories included "jx_results" as the main folder, which contained subfolders for "energy", "energy_filtered", "power", and "power_filtered". Each of these subfolders further contained "methods" and "calltrees" directories.

Next, the script created a directory named "mandelbrot_bitmap". The script then entered a loop where it ran the Java program with different parameters (15000, 20000, 30000, and 40000) for a certain number of iterations (30 in this case). The program was run with the JoularJX agent attached, which collected energy and power consumption data. For the first iteration, the program's output was saved as "mandelbrot_bitmap/java_temp.pmb".

After each execution, four types of files were created by JoularJX, representing energy and power data at different levels of granularity. These files had specific names based on the process ID of the Java program. The script then moved the non-empty files to their respective directories under "jx_results". If a file was empty, it was deleted.

Once the iterations were completed, the script proceeded to execute Python scripts for further data processing and analysis. The script "jx_gatherData.py" read energy and power data from CSV files in specific directories and created Pandas dataframes to store the data. The dataframes were then processed to extract relevant information (such as method names, parameters, iterations, and energy/power consumption), and this information was stored in lists. The lists were then used to create new dataframes with meaningful column names, which were then saved to CSV files. After that, the "jx_process_level_methods.py" script read data from a CSV file containing energy and power consumption values for different iterations of a process. It then calculated the total and average energy consumption, and total and average power consumption for each iteration, and wrote these results to a new CSV file. The code used the Python CSV module to read and write CSV files and stored the results in dictionaries. The "jx_process_level_methods.py" script analyzed the power consumption data at the method level, calculating energy and power for each method. The "jx_plot.py" script generated graphs based on the gathered data. Subsequently, the "jx_plot_geom_boxplot.py" script created box plots to visualize the total energy consumption across all process IDs.

Lastly, the script ran the "shapiro_wilk_test_energy.py" and "shapiro_wilk_test_power.py" scripts, which performed ShapiroWilk tests to check for normality in the energy and power consumption data, respectively.

**Experimental Procedure:**

To execute the experiment, perform the following steps:

1. **Clone the Repository:** Start by cloning the repository using the following command:

   ```
   git clone https://gitlabev.imtbs-tsp.eu/sticamsud/enact-internship-2023.git
   ```

2. **Navigate to the Project Directory:** Change the directory to the project folder:

   ```
   cd enact-internship-2023/Code/Prem_Experiment
   ```

3. **Install JoularJX:** Follow the instructions provided on the JoularJX website to install the JoularJX. Visit: https://github.com/joular/joularjx

4. **Compile the Java Program:** Compile the mandelbrot.java program by running the following command:

   ```
   javac mandelbrot.java
   ```

5. **Execute the Script:** Run the script by entering the following command:

   ```
   ./jx_script.sh mandelbrot
   ```

**Experimental Results and Analysis:**

In this section, we will examine the experimental results and analysis of the Shapiro-Wilk test conducted for both energy and power consumption of single java program(mandelbrot). Figure 4.1 presents the results of the test for energy, while Figure 4.2 shows the results for power. These figures help us understand the distribution of energy and power data. Moving on, we will also discuss the insights derived from the graphs representing total energy consumption (Figure 4.3) and total power consumption (Figure 4.4). These graphs provide a visual representation of the overall energy and power usage patterns. Finally, we will explore the Boxplot analysis of the total energy consumption, as depicted in Figure 4.5. This Boxplot allows us to identify any outliers and gain a better understanding of the spread and distribution of energy consumption data. Overall, these experimental results and analyses help us draw meaningful conclusions about the energy and power patterns observed in the experiment.

```
---- Shapiro-Wilk Test Results ----
Test Statistic: 0.8429595232009888
P-value: 5.870311459155175e-10
Conclusion: The data does not follow
a normal distribution(reject null hypothesis)
```

Figure 4.1: Shapiro-Wilk test results for Energy

```
---- Shapiro-Wilk Test Results ----
Test Statistic: 0.7128838896751404
P-value: 5.2058350170783654e-14
Conclusion: The data does not follow
a normal distribution(reject null hypothesis)
```

Figure 4.2: Shapiro-Wilk test results for Power

When the p-value obtained from the Shapiro-Wilk test is higher than the chosen significance level, it means that we do not have enough evidence to reject the null hypothesis of normality. That's mean that the data is approximately normally distributed. On the other hand, if the p-value is lower than the significance level, it indicates that there is evidence suggesting the data deviates from a normal distribution.

Based on the Shapiro-Wilk test we performed for "energy" (figure 4.1) and "power" (figure 4.2), the test statistic resulted in a value of 0.8429595232009888, 0.7128838896751404 and the associated p-value was found to be 5.870311459155175e-10, 5.2058350170783654e-14.

The null hypothesis in this case is that the data follows a normal distribution. However, with such a small p-value (below the conventional significance level of 0.05), it rejected the null hypothesis. Therefore, based on this test, the data for "energy" (figure 4.1) and "power" (figure 4.2) does not follow a normal distribution.

The conclusion made in the non-normality of data can be also be seen in their graph the data are not randomly distributed.

Figure 4.3 and Figure 4.4 present graphical representations of the total energy consumption and power consumption data, respectively. In Figure 4.3, the relationship between the Process ID (PID) and the corresponding energy usage is illustrated. The x-axis displays the unique Process IDs generated by the operating system during the runtime of each executed Java process, serving as identifiers

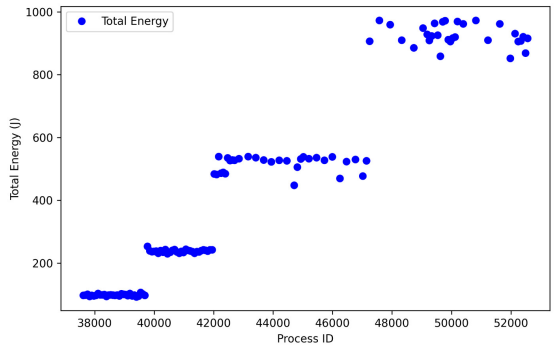


Figure 4.3: Graph of total energy consumption

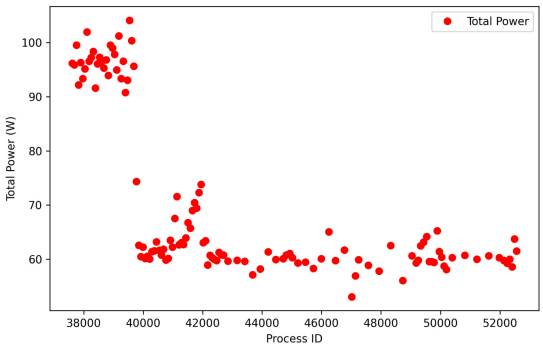


Figure 4.4: Graph of total power consumption

for process tracking and management. On the other hand, the y-axis represents the total energy consumption associated with each specific Process ID. Similarly, in Figure 4.4, the relationship between the Process ID (PID) and the corresponding power usage is depicted. The x-axis represents the unique Process IDs, while the y-axis denotes the total power consumption associated with each specific Process ID.

Upon examining the graphs, it is evident that there are four distinct clusters. These clusters can be attributed to the utilization of four parameters in the script, with values of 15,000, 20,000, 30,000, and 40,000. Each iteration, ranging from 1 to 30, contributes to the formation of these distinct clusters, enabling a comprehensive analysis of the energy and power consumption patterns. It is important to note that power consumption is proportional to energy consumption, and therefore, the clusters observed in the energy graph correspond to similar clusters in the power graph.

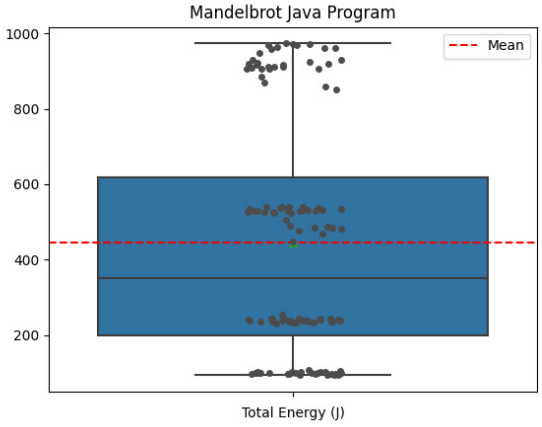


Figure 4.5: Boxplot of total energy consumption

The box plot(figure 4.5) analysis of the "Mandelbrot Java Program" provides valuable insights into the distribution and central tendency of its total energy consumption. The rectangular box represents the interquartile range, capturing the middle 50% of the data, while the median line denotes the central value. By comparing the mean, depicted as a red dashed line, with the median, we can assess the skewness of the distribution. If the mean and median align closely, it suggests a symmetrical

distribution; otherwise, skewness towards higher or lower values becomes apparent. The whiskers illustrate the range of typical values, excluding outliers that may indicate extreme energy consumption instances. By examining these outliers, we can identify exceptional cases where energy consumption significantly deviates from the norm. Overall, this box plot analysis offers valuable information on the distribution, central tendency, and potential outliers, enabling us to evaluate the energy efficiency of the Mandelbrot Java Program.

As we can see in the figure 4.5 that the mean is slightly far from the median and stays on the upper side of the median suggests that the distribution of the total energy consumption for the Mandelbrot Java Program is right-skewed. Right-skewness indicates that there are some instances of relatively higher energy consumption that are pulling the mean towards that direction.

The whiskers, which represent the range of typical values, touching the endpoints suggest that there are data points that lie at the extremes of the distribution. These points may be considered outliers or extreme values, indicating cases of significantly higher energy consumption compared to the rest of the data. This information help to assess the energy consumption of the Mandelbrot Java Program.

### 4.1.2 Energy consumption monitoring of a full project (cMath_original) using JoularJX:

In the second experiment, the energy consumption of a project called 'cMath_original' was monitored using JoularJX. Similar to the previous experiment, JoularJX version 2.0 was used to obtain updated tree structure results in the output directory. To execute all test cases of the project together, a new test class named 'RunAllSuite.java' was created in the test directory. To execute the 'RunAllSuite.java' class with JoularJX, it was necessary to download the cpsuite-1.2.6.jar.zip file and unzip it. A script called 'jx_script_math.sh' in the 'cMath_original' directory was used to execute the 'RunAllSuite.java' class, which captured power and energy measurements using the JoularJX and managed the resulting files. The script included several Python files: "jx_gatherData.py," "jx_plot.py," "jx_process_level_methods.py," "shapiro_wilk_test_energy.py," and "shapiro_wilk_test_power.py."

The script set up directories for storing output files. These directories included "jx_results" as the main folder, which contained subfolders for "energy", "energy_filtered", "power", and "power_filtered". Each of these subfolders further contained "methods" and "calltrees" directories.

The script set the Java classpath and ran the 'RunAllSuite.java' program in a loop for 30 iterations. During the first iteration, the output was saved to a file named 'java_temp.pmb,' and for subsequent iterations, the output was discarded. After each execution, four types of files were created by JoularJX, representing energy and power data at different levels of granularity. These files had specific names based on the process ID of the Java program. The script then moved the non-empty files to their respective directories under "jx_results". If a file was empty, it was deleted.

Once the iterations were completed, the script proceeded to execute Python scripts for further data processing and analysis. The script "jx_gatherData.py" read energy and power data from CSV files in specific directories and created Pandas data frames to store the data. The data frames were then processed to extract relevant information (such as method names, parameters, iterations, and energy/power consumption), and this information was stored in lists. The lists were then used to create new data frames with meaningful column names, which were then saved to CSV files. After that, the "jx_process_level_methods.py" script read data from a CSV file containing energy and power consumption values for different iterations of a process. It then calculated the total and average energy consumption, and total and average power consumption for each process id, and wrote these results

to a new CSV file. The code used the Python CSV module to read and write CSV files and stored the results in dictionaries. The "jx_process_level_methods.py" script analyzed the power consumption data at the method level, calculating energy and power for each method. The "jx_plot.py" script generated graphs based on the gathered data. Subsequently, the "jx_plot_geom_boxplot.py" script created box plots to visualize the total energy consumption across all process IDs.

Lastly, the script ran the "shapiro_wilk_test_energy.py" and "shapiro_wilk_test_power.py" scripts, which performed ShapiroWilk tests to check for normality in the energy and power consumption data, respectively.

**Experimental Procedure:**

To execute the experiment, perform the following steps:

1. **Clone the Repository:** Start by cloning the repository using the following command:

   ```
   git clone https://gitlabev.imtbs-tsp.eu/sticamsud/enact-internship-2023.git
   ```

2. **Navigate to the Project Directory:** Change the directory to the project folder:

   ```
   cd enact-internship-2023/Code/Prem_Experiment/cMath_original
   ```

3. **Install JoularJX:** Follow the instructions provided on the JoularJX website to install the JoularJX. Visit: https://github.com/joular/joularjx

4. **Navigate to the RunAllSuite.java file:**

   ```
   cd src/test/java
   ```

5. **Compile the RunAllSuite.java file:** Compile the RunAllSuite.java program by running the following command:

   ```
   javac -cp /home/tareq/cpsuite-1.2.6.jar:/usr/share/java/junit4.jar RunAllSuite.java
   ```

6. **Move the compiled file(RunAllSuite.class) in the bin folder:** Move the compiled file (RunAllSuite.class) in the bin folder by running the following command

   ```
   cd ../../../../../
   cd Prem_Experiment/cMath_original/bin
   ```

7. **Execute the Script:** Run the script by entering the following command:

   ```
   cd ../
   ./jx_script_math.sh
   ```

**Experimental Results and Analysis:**

In this section, we will present the experimental results and analysis of the ShapiroWilk test conducted on the energy and power consumption data of the cMath_original project. Figure 4.6 illustrates the results of the ShapiroWilk test for energy, while Figure 4.7 displays the results for power. These figures provide valuable insights into the distribution of energy and power data. Additionally, we will analyze the graphs representing the total energy consumption (Figure 4.8) and total power consumption (Figure 4.9) to visually understand the overall patterns of energy and power usage. Furthermore, we will perform Boxplot analysis on the total energy consumption (Figure 4.10) to identify any outliers and gain a comprehensive understanding of the distribution and spread of energy consumption data. By examining these experimental results and conducting relevant analyses, we can draw meaningful conclusions regarding the observed energy and power patterns in this experiment.

```
---- Shapiro-Wilk Test Results ----
Test statistic: 0.9073889255523682
P-value: 0.01279442012310028
The data does not follow a normal
distribution(reject null hypothesis)
```

Figure 4.6: Shapiro-Wilk test results for Energy

```
---- Shapiro-Wilk Test Results ----
Test statistic: 0.8638900518417358
P-value: 0.0012285438133403659
The data does not follow a normal
distribution(reject null hypothesis)
```

Figure 4.7: Shapiro-Wilk test results for Power

The Shapiro-Wilk test was conducted on the "energy" (figure 4.6) and "power" (figure 4.7) data which was achieved from full project experiment(cMath_Original), aiming to assess their adherence to a normal distribution. The test yielded test statistics of 0.9073889255523682 and 0.8638900518417358, with associated p-values of 0.01279442012310028 and 0.0012285438133403659, respectively. The null hypothesis assumes that the data follows a normal distribution. However, due to the obtained p-values being below the conventional significance level of 0.05, the null hypothesis is rejected. Consequently, it can be concluded that the data for both "energy" and "power" do not conform to a normal distribution. This conclusion is further supported by a visual examination of the graphs, which display non-random distribution patterns.

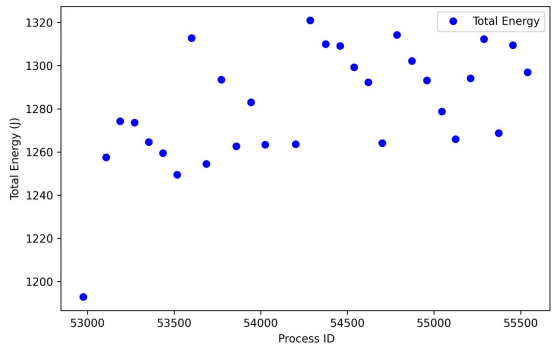


Figure 4.8: Graph of total energy consumption for cMath project

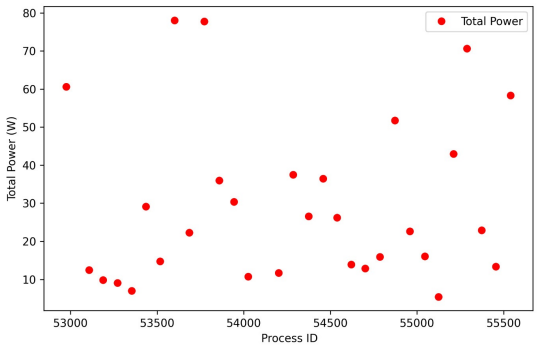


Figure 4.9: Graph of total power consumption for cMath project

Figures 4.8 and 4.9 illustrate the relationship between Process ID (PID) and energy consumption, as well as power consumption, respectively. In Figure 4.8, the x-axis represents unique Process IDs generated by the operating system for Java processes, while the y-axis displays the total energy

consumption associated with each PID. The graph shows energy consumption ranging from 1200 joules to 1320 joules. Notably, the first PID exhibits relatively low energy usage, while between PIDs 54000 and 54500, there is a significant spike in energy consumption exceeding the upper limit. This suggests a subset of processes with higher energy demands. On average, energy consumption falls between 1240 joules and 1320 joules, indicating overall consistent usage.

In Figure 4.9, the x-axis represents unique Process IDs, and the y-axis represents total power consumption. The graph shows power consumption ranging from 10 wards to 80 wards. Notably, between PIDs 53500 and 54000, there are two points where power consumption reaches approximately 80 wards, likely indicating processes or events requiring substantial power for intensive computational or operational activities. On average, power consumption falls within the range of 5 wards to 40 wards, suggesting a typical power requirement for most processes.

These graphical representations provide valuable insights into the patterns of energy and power consumption, highlighting specific PIDs with significantly higher energy and power demands. By analyzing these graphs, we can gain a better understanding of the energy and power requirements of various processes and identify areas where optimization or efficiency improvements may be necessary.

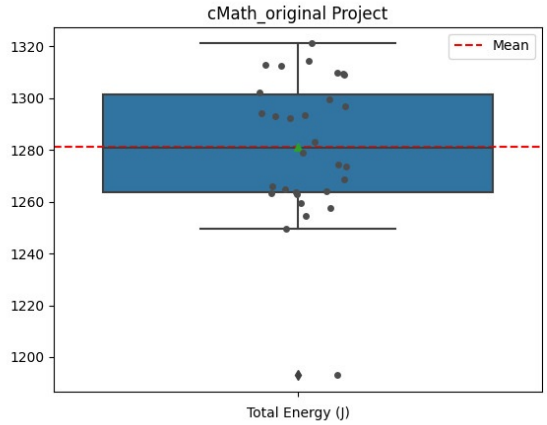


Figure 4.10: Boxplot of total energy consumption for cMath project

In the boxplot(figure 4.10) analysis of the energy consumption data for the full project (cMath_Original), we can observe the following: Firstly, the mean line, which represents the average energy consumption, is very close to the median line, indicating that the data is relatively symmetrically distributed. However, it is worth noting that the mean line is slightly higher than the median line, suggesting that there may be a few higher energy consumption values that are pulling the average up.

Additionally, two points are located beyond the whisker end lines, one on each side. This indicates the presence of outliers in the data, which are data points that deviate significantly from the overall pattern. These outliers could represent unusual or extreme energy consumption values.

In conclusion, the boxplot (figure 4.10) analysis provides valuable insights into the energy consumption of the full project (cMath_Original). The close proximity of the mean and median lines suggests a relatively symmetrical distribution, while the presence of outliers indicates the occurrence of unusual energy consumption values. By examining the range of the boxplot (figure 4.10), we can establish the typical energy usage range for the project. These findings serve as important information for assessing and managing the energy consumption of the project effectively.

## 4.2 Experimental executions using Gin

In this section, we will explain the conducted experimental executions using the Gin tool to obtain optimized or improved programs or projects. The focus of our experiment was based on a single criteria fitness function called "time execution". We performed two experiments: one on a single program, specifically running a simple example of a triangle, and the other on a full project called spatial4j, which involved a Maven project. To further enhance, we conducted two other experiments using joularJX. These experiments involved comparing the optimized and original versions of the single program and the full project. To determine if there were any significant differences, we utilized the Wilcoxon Test. After that, we considered multi criteria fitness function(time execution, memory consumption) based on this we optimised both the single program and the full project. Once again, we compared the original and optimized versions using the Wilcoxon Test to ascertain any notable distinctions.

Based on the results obtained, we integrated one code refactoring for energy saving into the Gin tool's . After the integration, we conducted further experiments to assess whether this integration had any impact on energy efficiency.

### 4.2.1 Running a Simple Example: Triangle

In this section, we will describe the experimental process we followed to optimize the "Triangle" Java program by optimizing the "classifyTriangle" method in the "Triangle" Java program using Gin's local search. Our objective was to obtain an optimized version of the program by focusing on a single criteria fitness function, namely "Time execution."

To begin the experiment, we selected the "Triangle" program as our target. The program likely computes the properties of triangles based on given inputs. We aimed to improve its performance by minimizing the time it takes to execute. The program was run with the specified command, and Gin's local search algorithm was applied.

**Experimental Procedure:**

To execute the experiment, perform the following steps:

1. **Prerequisites:** Gin requires:
   - JDK 17
   - Gradle (tested with version 8.0.2)
   - A number of dependencies, which can be downloaded manually or via Gradle (recommended)
   - For Maven projects: make sure the Java version is set to the same version as Gin's.
2. **Clone the Repository:** Start by cloning the repository using the following command:

   ```
   git clone https://github.com/gintool/gin.git
   ```

3. **Navigate to the examples Directory:** Change the directory to the project folder:

   ```
   cd gin/examples/triangle
   ```

4. **Compile the TriangleTest.java file:** Compile the TriangleTest.java program by running the following command:

   ```
   javac -cp /usr/share/java/junit4.jar:. TriangleTest.java
   ```

5. **Navigate to the gin Directory:** Change the directory to the gin folder:

   ```
   cd ../../
   ```

6. **Build using gradle (alternatively import into any IDE, such as IntelliJ) :**

   ```
   gradle build
   ```

   This will build and test Gin, and also create a fat jar at build/gin.jar containing all required dependencies.

   Note: If the provided build command will not work use the following command below:

   ```
   ./gradlew clean build -x test copyToLib
   ```

   This will ensure a clean build by removing any existing build artifacts, compiles the source code, skips test execution, and then executes the custom copyToLib task, which carries out additional project-specific actions defined in the build script and also create a fat jar at build/gin.jar containing all required dependencies.

7. **Execute (running Gin's local search on a simple example) by running the following command:**

   ```
   java -jar build/gin.jar -f examples/triangle/Triangle.java -m
   "classifyTriangle(int,int,int)"
   ```

   Note: If the provided command for Running a Simple Example will not able to work, follow the below informations and execute the commands:

   ```
   java -cp build/gin.jar:lib/junit-vintage-engine-5.9.2.jar gin.LocalSearch -f
   examples/triangle/Triangle.java -m "classifyTriangle(int,int,int)"
   ```

   Note: In this command specified the location of the necessary JAR files.

**Experimental Results and Analysis:**

This section presents the experimental results and analysis of the optimization experiment conducted to achieve an optimized Triangle Java program. The focus of the optimization was specifically on improving the "classifyTriangle" method within the program, utilizing the Gin tool.

```
2023-07-10 13:59:10 gin.LocalSearch.search() INFO: Localsearch on file: examples/triangle/Triangle.java method: classifyTriangle(int,int,int)
2023-07-10 13:59:27 gin.LocalSearch.search() INFO: Original execution time: 1606602189ns
2023-07-10 13:59:27 gin.LocalSearch.search() INFO: Step: 1, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":48 -> "examples/trian
gle/Triangle.java":8 |, Failed to compile
2023-07-10 13:59:27 gin.LocalSearch.search() INFO: Step: 2, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":33 -> "examples/triangle
/Triangle.java":17 |, Failed to compile
2023-07-10 13:59:27 gin.LocalSearch.search() INFO: Step: 3, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":46 -> "examples/triangle
/Triangle.java":34 |, Failed to compile
2023-07-10 13:59:27 gin.LocalSearch.search() INFO: Step: 4, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":13 -> "examples/trian
gle/Triangle.java":39 |, Failed to compile
2023-07-10 13:59:28 gin.LocalSearch.search() INFO: Step: 5, Patch: | gin.edit.line.DeleteLine "examples/triangle/Triangle.java":28 |, Failed to pass a
ll tests
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 6, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":21 -> "examples/triangle
/Triangle.java":13 |, Failed to pass all tests
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 7, Patch: | gin.edit.line.SwapLine "examples/triangle/Triangle.java":13 <-> "examples/triangl
e/Triangle.java":15 |, Failed to compile
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 8, Patch: | gin.edit.line.DeleteLine "examples/triangle/Triangle.java":35 |, Failed to compil
e
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 9, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":34 -> "examples/triangle
/Triangle.java":13 |, Failed to compile
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 10, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":6 -> "examples/triangle
/Triangle.java":27 |, Failed to compile
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 11, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":5 -> "examples/triangle
/Triangle.java":10 |, Failed to compile
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 12, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":33 -> "examples/triangl
e/Triangle.java":26 |, Failed to compile
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 13, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":35 -> "examples/tria
ngle/Triangle.java":32 |, Failed to compile
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 14, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":4 -> "examples/triangle
/Triangle.java":34 |, Failed to compile
2023-07-10 13:59:30 gin.LocalSearch.search() INFO: Step: 15, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":43 -> "examples/tria
ngle/Triangle.java":16 |, Failed to compile
2023-07-10 13:59:31 gin.LocalSearch.search() INFO: Step: 16, Patch: | gin.edit.line.DeleteLine "examples/triangle/Triangle.java":21 |, Failed to pass
all tests
2023-07-10 13:59:33 gin.LocalSearch.search() INFO: Step: 17, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":23 -> "examples/tria
ngle/Triangle.java":41 |, New best time: 1606376261(ns)
2023-07-10 13:59:35 gin.LocalSearch.search() INFO: Step: 18, Patch: |, New best time: 1605845707(ns)
```

Figure 4.11: Command line output: Optimized Triangle Java program with Gin tool.

The output(figure 4.11) of the program provided information about each step of the local search process. It started with a summary of the file and method being analyzed, followed by reporting the original execution time of the method.

For each step, the output indicated the patch being applied to the code. If a patch failed to compile, it meant that the modified code could not be compiled successfully. If the patched code was compiled, the runtime of the modified code was shown. Additionally, if the patched code passed all provided tests, the runtime was highlighted if it was the fastest successful result seen so far.

```
2023-07-10 14:00:13 gin.LocalSearch.search() INFO: Step: 95, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":16 -> "examples/tria
ngle/Triangle.java":28 |, Failed to pass all tests
2023-07-10 14:00:14 gin.LocalSearch.search() INFO: Step: 96, Patch: | gin.edit.line.SwapLine "examples/triangle/Triangle.java":15 <-> "examples/triang
le/Triangle.java":32 |, Failed to compile
2023-07-10 14:00:14 gin.LocalSearch.search() INFO: Step: 97, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":17 -> "examples/triangl
e/Triangle.java":22 |, Failed to compile
2023-07-10 14:00:14 gin.LocalSearch.search() INFO: Step: 98, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":29 -> "examples/tria
ngle/Triangle.java":26 |, Failed to compile
2023-07-10 14:00:15 gin.LocalSearch.search() INFO: Step: 99, Patch: | gin.edit.line.DeleteLine "examples/triangle/Triangle.java":27 |, Failed to pass
all tests
2023-07-10 14:00:15 gin.LocalSearch.search() INFO: Step: 100, Patch: | gin.edit.line.SwapLine "examples/triangle/Triangle.java":27 <-> "examples/trian
gle/Triangle.java":10 |, Failed to pass all tests
2023-07-10 14:00:15 gin.LocalSearch.search() INFO: Finished. Best time: 1605243842 (ns), Speedup (%): 0.08, Patch: |
```

Figure 4.12: Command line output: Optimized Triangle Java program with Gin tool.

The local search algorithm continued for a specified number of steps, which was set to 100 by default (figure 4.12). Throughout the process, different patches were applied and evaluated, aiming to find an optimized version of the "classifyTriangle" method for optimising the "Triangle" single program.

By examining the output(figure 4.12), we could able to identify the patches that resulted in successfully compiled code and improved runtime performance. The goal was to find the fastest code that passed all provided tests. At the end of the process, a brief summary was provided to give an overview of the optimization results.


Figure 4.13: Optimised program file of Triangle

In Figure 4.13, the directory where the optimized code was generated is specified. The figure displays the Optimised program file of the Triangle program in blue, representing the code that has been improved for better performance. Conversely, the red color represents the original program file of the Triangle program, which is the code in its initial state without any optimization. This color

distinction helps visually differentiate between the optimized and original versions of the code.

### 4.2.2 Full Example with a Maven Project: spatial4j

In this section, we will explain the conducted experimental executions using the Gin tool to obtain optimized or improved projects. Our focus was on optimizing the "time execution" of the selected project. We performed an experiment on a full project called spatial4j, which involved a Maven project for getting an optimized spatial4j project.

**Experimental Procedure:**

To execute the experiment, perform the following steps:

**Experimental Results and Analysis:**

### 4.2.3 Experimental executions using JoularJX for patchs given by Gin

| **Example** | **Execution Time (ns)** | **Energy Consumption** | **Memory Consumption (Mbytes)** | **Energy Consumption** | **Execution Time and Memory Consumption** | **Energy Consumption** |
|---|---|---|---|---|---|---|
| Original Triangle | 1636120217 ns | Mean: 5.833529999999999<br>SD: 1.1850097808849949 | 28 Mbytes | Mean: 7.241156666666667<br>SD: 0.7305546191119096 | 1773092415 ns<br>30 Mbytes | Mean: 8.28191<br>SD: 1.3588547404923825 |
| Optimised Triangle | 1616236347 ns<br>Speedup (%): 1.22 | Mean: 5.02284<br>SD: 1.1893153868420734 | 22 Mbytes<br>Memory reduction: 21.43% | Mean: 6.051069999999999<br>SD: 0.8395743569173029 | 1751420953 ns<br>Speedup (%): 1.22.<br>26 Mbytes<br>Memory reduction: 13.33% | Mean: 5.954906666666665<br>SD: 0.6897867226271056 |
| Original GCD | 2813479316 ns | Mean: 27.76620666666667<br>SD: 2.3485025482771698 | 180 Mbytes | Mean: 26.35339666666666<br>SD: 2.1860810185452126 | 2892005383 ns<br>158 Mbytes | Mean: 26.74704333333333<br>SD: 2.010952294172285 |
| Optimised GCD | 2462227777 ns<br>Speedup (%): 12.48 | Mean: 13.551136666666668<br>SD: 1.198982988428205 | 28 Mbytes<br>Memory reduction: 84.44% | Mean: 12.848373333333335<br>SD: 1.1839863707665377 | 2631710897 ns<br>Speedup (%): 9.00.<br>23 Mbytes<br>Memory reduction: 85.44% | Mean: 12.964123333333335<br>SD: 1.0991211252894169 |
| Original Rectangle | 2629499616 ns | Mean: 6.587116666666668<br>SD: 1.499966641027527 | 26 Mbytes | Mean: 6.43129<br>SD: 1.7408080134097856 | 2707476462 ns<br>31 Mbytes | Mean: 7.4105<br>SD: 1.2968423914554388 |
| Optimised Rectangle | 1213180874 ns<br>Speedup (%): 53.86 | Mean: 5.217436666666667<br>SD: 0.8279971778576481 | 14 Mbytes<br>Memory reduction: 46.15% | Mean: 5.299950000000001<br>SD: 1.594310723781759 | 1286266785 ns<br>Speedup (%): 52.59.<br>20 Mbytes<br>Memory reduction: 35.48% | Mean: 6.3404<br>SD: 1.4073167988814013 |

Table 4.1: Comparison between Single criteria fitness(Execution Time, Memory Consumption individually) and Multi criteria fitness(Execution Time, Memory Consumption combine).

## 4.3 Integrating code refactoring to Gin

# Chapter 5

# Conclusion

## 5.1 Conclusion

# Chapter 6

# Future Work

## 6.1 Future Work

# Bibliography

[Barack and Huang, 2018] Barack, O. and Huang, L. (2018). Effectiveness of code refactoring techniques for energy consumption in a mobile environment. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, pages 165–171. The Steering Committee of The World Congress in Computer Science, Computer .... 18

[Bekaroo et al., 2014] Bekaroo, G., Bokhoree, C., and Pattinson, C. (2014). Power measurement of computers: analysis of the effectiveness of the software based approach. *Int. J. Emerg. Technol. Adv. Eng*, 4(5):755–762. 13

[Brownlee et al., 2019] Brownlee, A. E. I., Petke, J., Alexander, B., Barr, E. T., Wagner, M., and White, D. R. (2019). Gin: genetic improvement research made easy. In Auger, A. and Stützle, T., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 985–993. ACM. 5, 19, 20, 21, 22, 24

[Bruce et al., 2015] Bruce, B. R., Petke, J., and Harman, M. (2015). Reducing energy consumption using genetic improvement. In Silva, S. and Esparcia-Alcázar, A. I., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, pages 1327–1334. ACM. 20

[Burles et al., 2015] Burles, N., Bowles, E., Brownlee, A. E. I., Kocsis, Z. A., Swan, J., and Veerapen, N. (2015). Object-oriented genetic improvement for improved energy consumption in google guava. In de Oliveira Barros, M. and Labiche, Y., editors, *Search-Based Software Engineering - 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*, volume 9275 of *Lecture Notes in Computer Science*, pages 255–261. Springer. 20

[Callan and Petke, 2022] Callan, J. and Petke, J. (2022). Multi-objective genetic improvement: A case study with evosuite. In Papadakis, M. and Vergilio, S. R., editors, *Search-Based Software Engineering - 14th International Symposium, SSBSE 2022, Singapore, November 17-18, 2022, Proceedings*, volume 13711 of *Lecture Notes in Computer Science*, pages 111–117. Springer. 20

[Feitosa et al., 2017] Feitosa, D., Alders, R., Ampatzoglou, A., Avgeriou, P., and Nakagawa, E. Y. (2017). Investigating the effect of design patterns on energy consumption. *J. Softw. Evol. Process.*, 29(2). 12

[Kim et al., 2018a] Kim, D., Hong, J., Yoon, I., and Lee, S. (2018a). Code refactoring techniques for reducing energy consumption in embedded computing environment. *Clust. Comput.*, 21(1):1079–1095. 13, 16

[Kim et al., 2018b] Kim, D., Hong, J.-E., Yoon, I., and Lee, S.-H. (2018b). Code refactoring techniques for reducing energy consumption in embedded computing environment. *Cluster computing*, 21:1079–1095. 18

[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 Trans. Software Eng.*, 44(12):1176–1206. 17, 18

[Mrazek et al., 2015] Mrazek, V., Vasícek, Z., and Sekanina, L. (2015). Evolutionary approximation of software for embedded systems: Median function. In Silva, S. and Esparcia-Alcázar, A. I., editors, *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, pages 795–801. ACM. 20

[Noureddine, 2022] Noureddine, A. (2022). Powerjoular and joularjx: Multi-platform software power monitoring tools. In *18th International Conference on Intelligent Environments, IE 2022, Biarritz, France, June 20-23, 2022*, pages 1–4. IEEE. 13, 14

[Ournani et al., 2021] Ournani, Z., Rouvoy, R., Rust, P., and Penhoat, J. (2021). Tales from the code #2: A detailed assessment of code refactoring's impact on energy consumption. In Fill, H., van Sinderen, M., and Maciaszek, L. A., editors, *Software Technologies - 16th International Conference, ICSOFT 2021, Virtual Event, July 6-8, 2021, Revised Selected Papers*, volume 1622 of *Communications in Computer and Information Science*, pages 94–116. Springer. 17

[Palomba et al., 2019] Palomba, F., Nucci, D. D., Panichella, A., Zaidman, A., and Lucia, A. D. (2019). On the impact of code smells on the energy consumption of mobile applications. *Inf. Softw. Technol.*, 105:43–55. 17

[Park et al., 2014] Park, J. J., Hong, J., and Lee, S. (2014). Investigation for software power consumption of code refactoring techniques. In Reformat, M. Z., editor, *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013*, pages 717–722. Knowledge Systems Institute Graduate School. 18

[Petke et al., 2018] Petke, J., Haraldsson, S. O., Harman, M., Langdon, W. B., White, D. R., and Woodward, J. R. (2018). Genetic improvement of software: A comprehensive survey. *IEEE Trans. Evol. Comput.*, 22(3):415–432. 19

[Sahin et al., 2014] Sahin, C., Pollock, L. L., and Clause, J. (2014). How do code refactorings affect energy usage? In Morisio, M., Dybå, T., and Torchiano, M., editors, *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, pages 36:1–36:10. ACM. 17, 18

[Şanlıalp et al., 2022] Şanlıalp, İ., Öztürk, M. M., and Yiğit, T. (2022). Energy efficiency analysis of code refactoring techniques for green and sustainable software in portable devices. *Electronics*, 11(3):442. 12, 13, 18

[Schaarschmidt et al., 2020] Schaarschmidt, M., Uelschen, M., Pulvermüller, E., and Westerkamp, C. (2020). Energy-aware pattern framework: The energy-efficiency challenge for embedded systems from a software design perspective. In Ali, R., Kaindl, H., and Maciaszek, L. A., editors, *Evaluation of Novel Approaches to Software Engineering - 15th International Conference, ENASE 2020, Prague, Czech Republic, May 5-6, 2020, Revised Selected Papers*, volume 1375 of *Communications in Computer and Information Science*, pages 182–207. Springer. 11, 12

[Treibig et al., 2010] Treibig, J., Hager, G., and Wellein, G. (2010). LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. *CoRR*, abs/1004.4431. 13, 14

[University et al., 2016] University, R., of Bristol, U., Institute, I. S., and Limited, X. (2016). Entra 318337 — whole-systems energy transparency — energy-aware software development methods and tools. Technical report, IMDEA Software Institute, Madrid, Spain. 12

[Vos et al., 2022] Vos, S., Lago, P., Verdecchia, R., and Heitlager, I. (2022). Architectural tactics to optimize software for energy efficiency in the public cloud. In *International Conference on ICT for Sustainability, ICT4S 2022, Plovdiv, Bulgaria, June 13-17, 2022*, pages 77–87. IEEE. 11, 12

[Zuo et al., 2022] Zuo, S., Blot, A., and Petke, J. (2022). Evaluation of genetic improvement tools for improvement of non-functional properties of software. In Fieldsend, J. E. and Wagner, M., editors, *GECCO '22: Genetic and Evolutionary Computation Conference, Companion Volume, Boston, Massachusetts, USA, July 9 - 13, 2022*, pages 1956–1965. ACM. 19, 20