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 undertaken to explore tactics for enhancing software energy efficiency. We have chosen code refactoring as a tactic to improve energy efficiency in software. Subsequently, we utilized the genetic improvement(gin) tool to obtain an optimized version of the code. We then examined whether this optimized version results in a reduction in energy consumption. Our findings confirmed that the optimized program does indeed consume less energy. Our next step is to integrate code refactoring techniques with the GIN tool. Finally, we will conduct experiments to determine if the integrated gin tool can indeed bring about a significant reduction in energy consumption. 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 the foundation for the study. Existing approaches, techniques, and technologies related to software energy efficiency were analyzed, enabling an 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 Ecole des Mines de Saint-Étienne and Jean Monnet University. Their dedication and investment in 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 Context and Motivation

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) continues to heavily rely on fossil fuels[2].

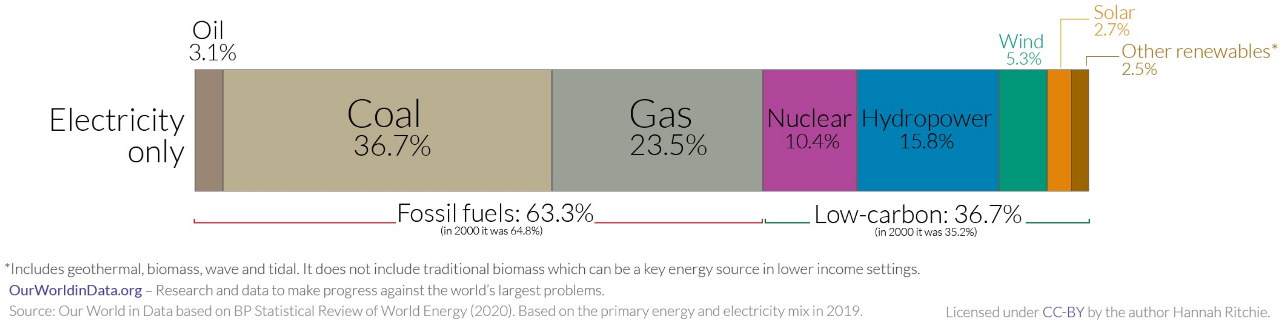


Figure 1.1: Global electricity production from fossil fuels[3].

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 (e;g.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].

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

[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.png
[4] https://www.theguardian.com/environment/2017/dec/11/tsunami-of-data-could-consume-fifth-global-electricity-by-2025
[5] https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

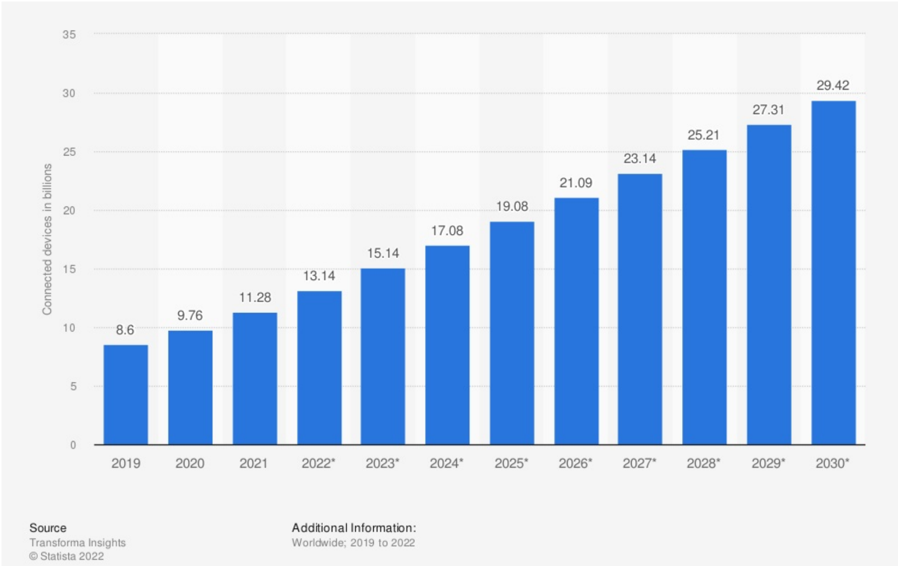


Figure 1.2: Number of IOT connected devices worldwide (2019-2021) with forecasts to 2030.[5]

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.

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[6].

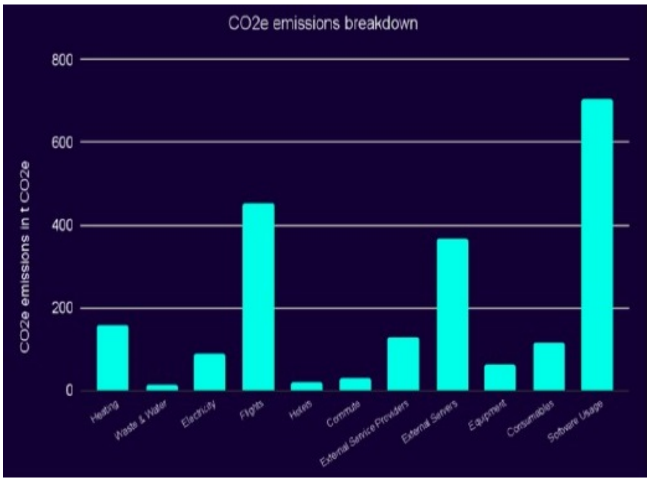


Figure 1.3: CO_2e emissions breakdown.[7]

[6] https://www.polygon.com/features/22914488/video-games-climate-change-carbon-footprint
[7] https://www.planetly.com/articles/what-tech-companies-can-do-to-reduce-and-avoid-emissions

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[8]. 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. For example: Puzzling out Software Sustainability [Calero and Piattini, 2017].

Energy efficiency is crucial in Cyber-Physical Social Systems (CPSS), which combine computing, physical processes, and social interactions. As these systems grow in complexity, they use more energy, impacting the environment. Making the software in CPSS more energy-efficient can cut down its overall energy use and reduce its carbon footprint. This not only helps the planet but also makes the system run better and be more resilient. The unique blend of tech and social elements in CPSS offers both challenges and opportunities for making it more energy-efficient. Thus, focusing on energy-smart software is key to fighting global climate change.

1.2 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[9]. 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++. For example: Ranking programming languages by energy efficiency [Pereira et al., 2021]. 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.

1.3 Objectives

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 into their

[8] https://hbr.org/2020/09/how-green-is-your-software

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

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

In RQ2, we focus on automating the integration of tactics to lower energy consumption. The approach involves using a genetic improvement, 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.

1.4 Conclusion

The efficient utilization of smart devices and the transition towards sustainable energy sources are crucial topics of global significance. It is essential for individuals and communities to remain cognizant of their electricity consumption and to promote energy efficiency. By raising awareness and making small changes in our daily lives, we can collectively contribute to reducing the strain on non-renewable resources and pave the way for a greener, more sustainable future.

The research comprised several distinct chapters. In the first chapter, the motivation behind the study, the objectives to be achieved, and the problem statement were discussed. In the second chapter, we delved into tactics for enhancing software energy efficiency, analyzed their respective benefits and limitations, and identified a tactic. Additionally, we emphasized the importance of energy consumption monitoring and selected *JoularJX* as our primary tool for monitoring energy consumption in Java-based applications. The third chapter provided an in-depth literature review focusing on the themes of code refactoring and genetic improvement in software. Chapter four detailed a preliminary study aimed at understanding software energy consumption using the *JoularJX* tool, illustrating its installation process and the initial experiments that were conducted using this tool. In Chapter five, we presented the results of our experiments, compared the energy consumption of the original and optimized programs using the *Gin* tool, and discussed the implications for software energy efficiency. The sixth chapter was for the conclusion and discussions on future work.

Chapter 2

Background

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

RQ1: Which tactics help to improve Energy Efficiency?

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, Design Patterns, and Code Refactoring.

- **Architectural Tactics**:These tactics focus on adapting the software architecture for energy efficiency. For example, [Paradis et al., 2021] provide a basis for reasoning about design decisions for energy efficiency by deriving a set of reusable architectural tactics derived from the research literature, via a taxonomic literature review. Researchers used an open-search and snowballing methodology to obtain primary studies and then used thematic coding to identify commonalities among the design strategies described. The result of this process is a taxonomy of 10 architectural tactics for energy efficiency, which provide a rational basis for architectural design and analysis for energy efficiency. These tactics are grouped into three broad categories: Resource Monitoring, Resource Allocation, and Resource Adaptation. These categories serve as a high-level checklist for a software architect or a reviewer, and the true design thinking goes into how those categories are refined into specific tactics and how those tactics are in turn translated into code, patterns, and components.

- **Design Patterns**:These are reusable solutions to common problems in software design that can be used to improve energy efficiency. For example, [Noureddine and Rajan, 2015] presents a vision to automatically detect and transform design patterns during compilation for better energy efficiency without impacting existing coding practices. The authors propose compiler transformations for two design patterns, Observer and Decorator, and perform an initial evaluation of their energy efficiency.

- **Code Refactoring**:This is the process of restructuring existing code without changing its external behavior(i.e. functionalities) to improve its readability and maintainability. Refactoring techniques aim to reduce the energy consumption of the software. [Şanlıalp et al., 2022] examines the effect of code refactoring techniques(e.g. Encapsulate field, Inline temp, Simplify nested loop) 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 the mentioned tactics. This is not an exhaustive study but it provides a general overview of advantages and limitations of these techniques:

Tactic	**Purpose**	**Advantages**	**Limitations**	**Example of reference**
Architectural Tactics	provide a basis for reasoning about design decisions for energy efficiency in software architectures. The paper derives a set of reusable architectural tactics for energy efficiency from the research literature, via a taxonomic literature review. Researchers used an open-search and snowballing methodology to obtain primary studies, and then used thematic coding to identify commonalities among the design strategies described. The result of this process is a taxonomy of 10 architectural tactics for energy efficiency. These tactics provide a rational basis for architectural design and analysis for energy efficiency.	Provide a rational basis for architectural design and analysis for energy efficiency. By using mentioned tactics, software architects can make informed decisions about how to design their systems to be more energy-efficient. This can help reduce the environmental impact of software, as well as improve the battery life of mobile and IoT devices. Additionally, by using an architectural approach to energy efficiency, software engineers can better manage complex system-wide properties, which can be difficult to address through coding alone.	Need for a comprehensive framework that can enumerate relevant contextual factors and assist in reasoning about the consequences of design decisions on energy efficiency and other quality attributes. The absence of such a framework makes it difficult for architects and developers to make informed decisions.	[Paradis et al., 2021]
Design Patterns	Explore the ways to improve the energy efficiency of software design patterns while retaining their essential benefits, such as improved code readability and maintainability. In this study researchers propose compiler transformations for two design patterns, Observer and Decorator, and perform an initial evaluation of their energy efficiency. Their vision is to automatically detect and transform design patterns during compilation for better energy efficiency without impacting existing coding practices	Several advantages of the proposed approach to improving the energy efficiency of software design patterns include: Developer coding practices remain unaffected. Benefits of using design patterns are retained. Energy consumption of software is reduced.	The study focuses on only two patterns (Decorator and Observer), limiting generalization.The empirical evaluation is based on a small set of programs, potentially limiting applicability.Transformations for energy optimization are applied manually, needing automation for scalability.	[Noureddine and Rajan, 2015]

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.	Code refactoring for energy efficiency can be a complex and time-consuming process that requires a deep understanding of the software and its energy consumption characteristics.	[Şanlıalp et al., 2022] and [Kim et al., 2018a]

Table 2.1: Comparison between the mentioned tactics

Unlike other tactics, code refactoring 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.

2.1 Energy Consumption Profiling Tools

Measuring energy consumption 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**: Wattmetter [Bekaroo et al., 2014]
- **Software Tools**:
  - Power Joular [Noureddine, 2022]
  - JoularJX [Noureddine, 2022]
  - Likwid Powermeter[1] [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 description of all the aforementioned software tools.

PowerJoular: PowerJoular is a command line software to monitor, in real time, the power consumption of software and hardware components.

Advantages:

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

It can measure the CPU, GPU and memory consumption. It writes the power consumption in a CSV file.

Disadvantages:

Power Joular can only measure the power consumption of Intel RAPL (CPU) and NVIDIA SMI (GPU).

JoularJX: JoularJX is a Java-based agent for software power monitoring at the source code level

Advantages:

JoularJX works as a java agent. It hooks to the JVM (Java Virtual Machine) to monitor power consumption. It can get power and energy consumption at the method level.

Disadvantages:

JoularJX can only measure the energy and power consumption for the Java source codes and applications.

Likwid Powermeter: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.

Advantages:

It can monitor the energy consumption by the core of the CPU of the machine, provide the result by measuring the energy consumption by each processor.

Disadvantages:

It can not able to monitor the energy consumption methodwise. Provide processorwise results (at the level of CPU cores), sometimes results can vary as other processes may run on the same core with the test process.

From 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. 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 1.2, our primary focus is on Java programs or Java-based software for making energy-efficient, which makes JoularJX the appropriate choice for our requirements.

2.2 Conclusion

In our quest to enhance software energy efficiency, we explored various tactics, such as Architectural Tactics, Design Patterns, and Code Refactoring. Through comparative analysis, we discerned the benefits and limitations of each tactic, with code refactoring proving to be versatile across diverse software types.This provided the answer to our first research question **RQ1**. Monitoring energy consumption is pivotal to this research. Among the software tools considered for profiling energy consumption, JoularJX emerged as the most appropriate for our Java-based software focus.

Chapter 3

Literature Review

Conducting a literature review is an essential step in any research project. In this section, we reviewed research papers related to the internship's topics, including code refactoring for improving software energy efficiency and genetic improvement for getting better versions of software. Through this review, we assessed the strengths and limitations of the existing research.

The insights gained from this review were in identifying potential code refactoring techniques for improving software energy efficiency, this answering RQ2.

3.1 Code Refactoring for Software Energy Efficiency

Code refactoring is the process of restructuring existing computer code without changing its external behavior to enhance reusability and maintainability of software. 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, the main limitations of applying code refactoring is that could be time expensive and error-prone. So, its automatising is crucial for supporting developers and architects.

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 (see Table 3.1 that presents three main studies in this domain).

[Sahin et al., 2014] presented an empirical study to investigate the energy impacts of 197 applications using 6 commonly-used code refactoring methods, for instance `Convert Local Variable to Field` (see Row 1 in Table 3.1). 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.

[Morales et al., 2018] proposed a code smell or anti-pattern correction approach called EARMO. EARMO is a multi-objective search based technique that improves energy efficiency and the quality of code by using code refactoring. In this study, the researchers analyzed the impact of eight types of anti-patterns, for instance `Move Method` that refactors the `Bob (God class)` code smell (see Row 2 in Table 3.1). EARMO is able to remove a median of 84% of code smells or anti-patterns. Moreover, EARMO extended the battery life of a mobile phone by up to 29 minutes. An experiment with developer showed that EARMO suggest 68% of code refactoring are relevant for improving the quality of code and energy efficiency.

[Palomba et al., 2019] founded that code refactoring for android applications, for instance refactoring the `Data Transmission Without Compression` code smell (see Row 3 in Table 3.1), is able to reduced energy consumption by up to 10.8%. Moreover, four code smells increase method energy consumption by up to 87 times.

Reference	**Code refactoring**	**Experiment / Benchmarking**	**Tool for EC**	**Results**
[Sahin et al., 2014]	**Code refactoring (6)**: Convert Local Variable to Field, Extract Local Variable, Extract Method, Introduce Indirection, Inline Method Introduce Parameter Object	9 Java Applications (ex. cMath, cCollections, ..)	Low Power Energy Aware Process (LEAP)	-7.50% to 4.54%
[Morales et al., 2018]	**Android code smells (3)**: Binding resources too early class, Private getter and setters, HashMap usage. **OO code smells (5)**: Lazy class, Blob (God class), Long-parameter list, Refused bequest, Speculative Generality	**Phase 1:** Empirical Study to understand in which extent 8 code refactorings help to save energy. **Phase 2:** EARMO is developped to select optimal series of code refactoring. The energy consumed by the version of code is inferred from Phase 1.	Not reported.	EARMO is able to save 29 minutes of battery.
[Palomba et al., 2019]	**Android-specific code smells (9)**: Data Transmission Without Compression, Durable Wakelock, Inefficient Data Structure, Inefficient SQL Query, Inefficient Data Format And Parser, Internal Setter, Leaking Thread, Member-Ignoring Method, Slow Loop	60 Android Java apps (categories ex. games, productivity, social, etc)	PETRA (Power Estimation Tool for Android)	Four code smell types increase method energy consumption by up to 87 times.

Table 3.1: Comparison of approaches

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.

In Table 3.2, we present the code refactoring techniques studied in the previous works. We introduce them because the authors of these works provided individual information about their positive impact in energy efficiency. So, it provides us more insights about their usefulness in software energy efficiency.

Ref.	**Code Refactoring Technique**	**Purpose**	**Energy Consumption Impact**
[Sahin et al., 2014]	Convert Local Variable to Field	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	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	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	Copies the body of a called 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	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	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.
[Park et al., 2014]	Encapsulate Field	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.
[Barack and Huang, 2018]	Inline Temp	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.
[Morales et al., 2018]	Move Method for refactoring the Blob (God class) code smell	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.
[Kim et al., 2018b]	Simplify Nested Loop	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.

Table 3.2: Types of Code Refactoring Techniques

From Table 3.2, we can observe that `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. 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 code-base.

For integrating code refactoring techniques into the a genetic improvement method for improving software energy efficiency, we can consider the `Convert Local Variable to Field` and `Introduce Parameter Object` code refactoring techniques because they consistently yield energy reductions for Java programming. Moreover, the `Move Method` code refactoring technique can be considered for Java applications to enhance energy efficiency. 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.

3.2 Genetic Improvement (GI)

Genetic Improvement (GI) is a field of research that uses automated search to find improved versions of existing software. 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 [Petke et al., 2018, Zuo et al., 2022].

Genetic Improvement (GI) is an optimization technique inspired by natural evolution to enhance software. The process starts with an initialization phase where a set of software solutions (individuals) is created. Each solution's efficacy is gauged using a fitness function. If the stopping criteria are not met, which can be a specific fitness level or a number of iterations, the solutions undergo mutations to introduce variability. This altered set is then re-evaluated. The cycle of evaluation, checking stopping criteria, and mutation continues until the stopping criteria are met, at which point the best solution is identified as optimal. Throughout this process, GI aims to find a satisfactory software version based on predefined metrics, though a global optimum is not always guaranteed.

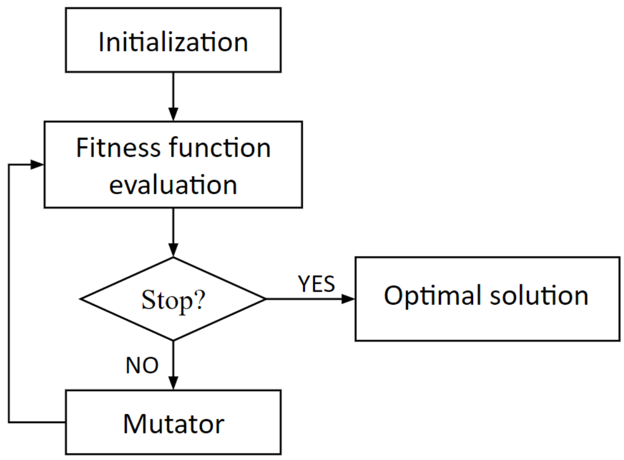


Figure 3.1: Overall process of genetic improvement

[Petke et al., 2018] presents a comprehensive survey of research studies on GI that were published between 1995 and 2015. Authors identified that 96% of these studies use evolutionary algorithms (in particular genetic programming). Moreover, GI has resulted in performance improvements for a diverse set of properties such as execution time and memory consumption, as well as results for fixing and extending existing system functionality. However, only three studies focused on reducing energy consumption were mentioned in this survey. These studies are described as follows:

- [Bruce et al., 2015] 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%.
- [Mrazek et al., 2015] 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.
- [Burles et al., 2015] 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.

After a thorough analysis of existing research work on GI, it becomes evident that 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.

[Zuo et al., 2022] 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. The study found 63 GI papers that introduce a GI tool to improve non-functional properties of software, out of which 31 are accompanied by open-source code. From these tools, the study was able to successfully run 8 GI tools. Among these 8 GI tools, only 2, *i.e.,* Gin and PyGGI, can be readily applied to new software for the improvement of non-functional properties. The Gin tool[1] provides an extensible and modifiable toolbox for GI experimentation, specifically targeting the Java ecosystem. The PyGGI tool[2] offers a Python General lightweight and simple framework for Genetic Improvement. In the subsequent section we will discuss about Gin tool.

As mentioned in Section 1.2 of Chapter 1, we selected the Java programming language as our primary focus for our research work because it spends less energy than other programming languages such as Python. Therefore, we selected the Gin tool, a Java-based Genetic Improvement tool, to help us improve non-functional properties of software with the objective of reducing energy consumption.

3.2.1 The Gin Toolbox

Gin [Brownlee et al., 2019] is a 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

[1] https://github.com/gintool/gin

[2] https://github.com/coinse/pyggi

of Java projects, Gin supports various aspects of software improvement, including program repair, run-time optimization, energy consumption reduction, and the addition of new functionality.

In a recent work [Callan and Petke, 2022], an extension of the Gin toolbox was presented. Gin was extended with a multi-objective search algorithm, *i.e.,* NSGA-II. The multi-objective extension of Gin was utilized to improve both the execution time and memory usage of EvoSuite as case study. 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.

Gin 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.

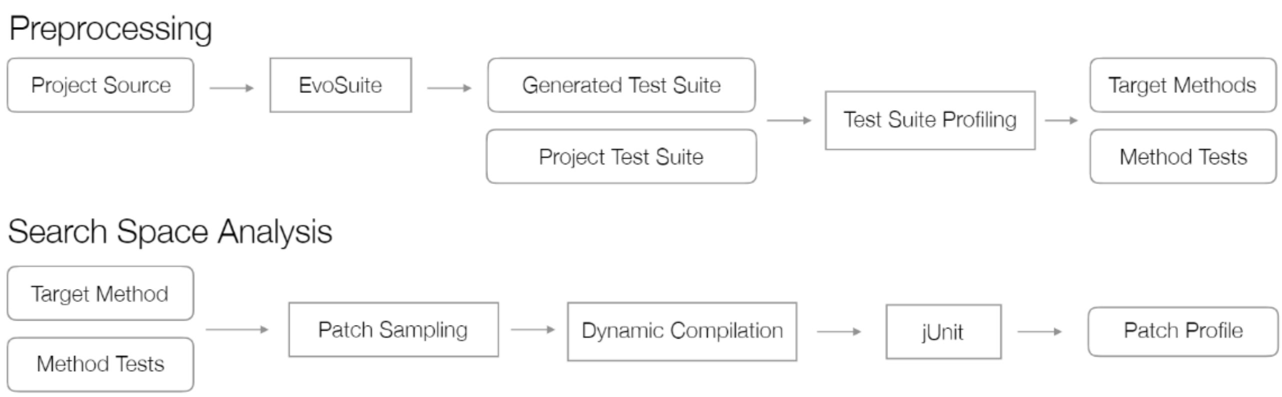


Figure 3.2: Gin Pipelines from [Brownlee et al., 2019]

As shown in Figure 3.2, Gin provides two pipelines: *Preprocessing* and *Search Space Analysis*:

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

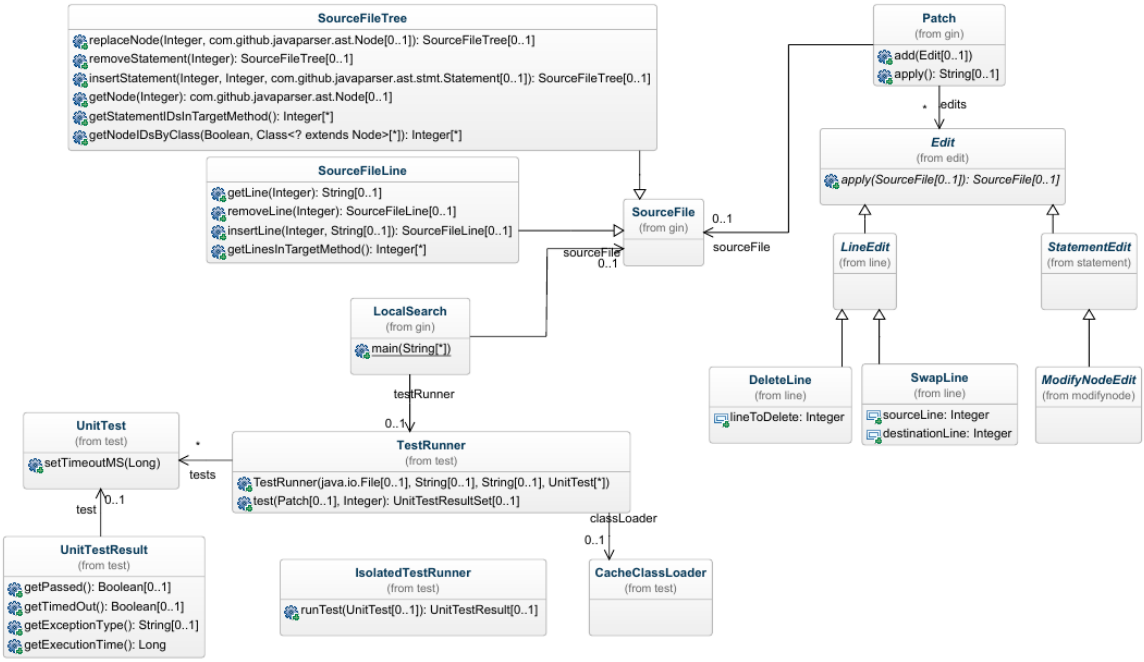


Figure 3.3: Gin Core Classes from [Brownlee et al., 2019]

The Gin toolkit encompasses a collection of classes mentioned in Figure 3.3 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.

3.2.2 Identify the element need to be extended in the Gin tool

In the previous subsection 3.2.1, 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.

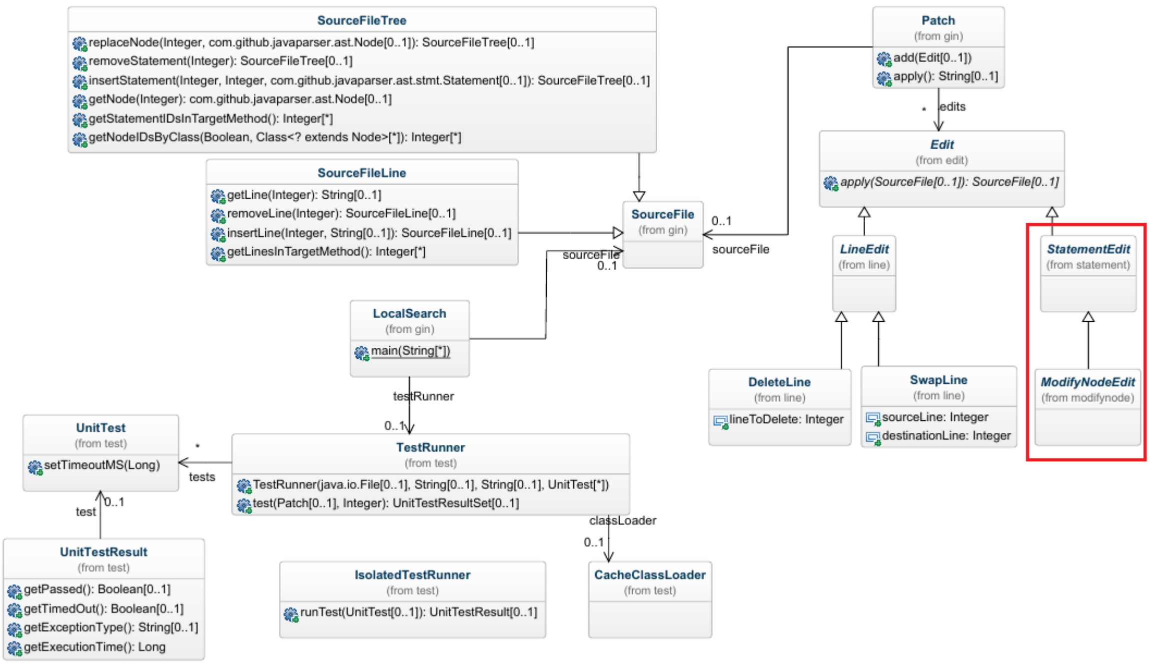


Figure 3.4: Extension of the Gin Tool's class. [Brownlee et al., 2019]

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

3.3 Conclusion

Through a comprehensive literature review, it has been observed that while various code refactoring techniques can improve software performance, their impact on energy efficiency varies based on the software's specific context. Notably, techniques like 'Convert Local Variable to Field', 'Introduce Parameter Object', and others have shown promise in reducing energy consumption. The Genetic Improvement (GIN) tool, which leverages Genetic Programming for software enhancement, presents an ideal platform to integrate these techniques. The StatementEdit class within GIN appears to be the most fitting for this integration, paving the way for potentially higher software quality and energy efficiency. This study effectively addresses the research question, highlighting the potential synergies between refactoring techniques and the GIN tool. 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.

Chapter 4

Preliminary study for empirical evaluations

Our main objective is to reduce energy consumption in software using architectural software tactics. In order to delve deeper into this subject, it is essential to gain a comprehensive understanding of how we can measure the energy consumption of a program or project.

As mentioned, *JoularJX* is a tool that can be used to monitor the energy consumption of a Java program or project at the method level. It can be hooked to the Java Virtual Machine when starting a Java program and provides real-time power consumption data for every method in the monitored program.

For our experiments, JoularJX was installed following the guidelines on `https://github.com/joular/joularjx`. We use version 2.0 of JoularJX, which requires a minimum of Java 11+ to run. On Windows, it depends on the Intel Power Gadget API, and on GNU/Linux, it uses the Intel RAPL interface through powercap.

All the experiment in this report were conducted on Dell Latitude 7490 laptop (Intel Core i7-8650U CPU @ 1.90GHz) running Debian GNU/Linux 11 (bullseye), Java 17 and JoularJX 2.0.

We conducted two preliminary experiments using JoularJX. These experiments are detailed as follows:

4.1 Energy consumption monitoring of a single Java program

In this experiment, the energy consumption of a single Java program, *i.e.,* `mandelbrot`[1] java program, was monitored using *JoularJX 2.0*. To do that, a bash script named `jx_script.sh` was created. The script created directories to organize the results and ran the Java program with different parameters for 30 iterations with the JoularJX agent attached. Then, it sequentially executes the following Python files (see Appendix A.1 for more details):

1. `jx_gatherData.py`: reads 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.

[1] `mandelbrot` was taken from the bench-marking used in [Pereira et al., 2017], [Couto et al., 2017]and [Lima et al., 2016] `http://benchmarksgame.alioth.debian.org/`

2. `jx_process_level_methods.py`: reads 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, 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.

3. `jx_plot.py`: generates graphs based on the gathered data. It creates box plots to visualize the total energy consumption and power consumption across all process IDs.

4. `shapiro_wilk_test_energy.py and shapiro_wilk_test_power.py`: performes `Shapiro-Wilk` tests to verfy the type of distribution, *i.e.,* normal or non- normal distributions, in the energy and power consumption data.

4.1.1 Experimental Results and Analysis

Figures 4.1 and 4.2 shows the Shapiro-Wilk test outcomes for the gathered energy and power consumption. As seen, the obtained were $\tilde{0}.843$ and $\tilde{0}.713$, with *p-values* of 5.87×10^{-10} and 5.206×10^{-14}, respectively. Given these confident *p-values* (less than 0.05), we can conclude that neither set of data follows a normal distribution.

```
---- 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(`mandelbrot` program)

```
---- 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(`mandelbrot` program)

Moreover, Figures 4.3 and 4.4 shows the box-plot for the data distribution of energy and power consumption of the processes related to the execution of the java program. These visual aids collectively facilitate a comprehensive interpretation of the energy and power behaviors observed. For instance, we can observe four distinct clusters for energy consumption. It corresponds to the script parameter values, we used 15 000, 20 000, 30 000, and 40 000 as parameters of the `mandelbrot` code. A greater value for the parameter, a higher energy consumption values. Because at least the time is extended. Seeing the graph related to power consumption there is a more stable use of CPU for executing the `mandelbrot` code. These clusters were formed over 30 iterations.

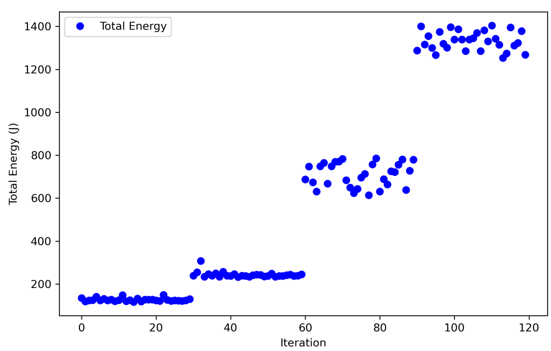


Figure 4.3: Graph of total energy consumption(`mandelbrot` program)

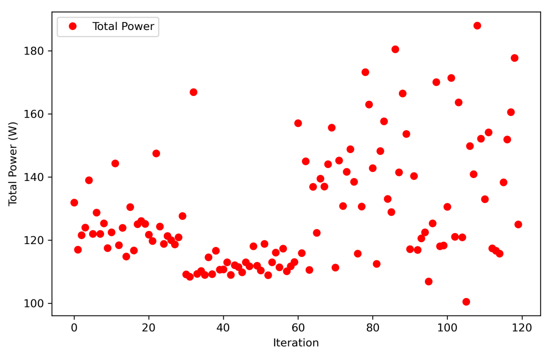


Figure 4.4: Graph of total power consumption(`mandelbrot` program)

4.2 Energy consumption monitoring of a Java project

In the second experiment, the energy consumption of a Java project, *i.e.,* `cMath`[2], was monitored using *JoularJX 2.0* as the previous experiment.

For a Java application, we execute its test cases classes that exercise the functionality of the Java application. The `cMath` application already contains a set of test cases classes. To execute all test cases of the project together, a new test class named `RunAllSuite.java` (see Appendix A.2 for more details) was created in the `test` directory. To execute the `RunAllSuite` class with JoularJX, it was necessary to download the `cpsuite-1.2.6`[3] jar.

A script called `jx_script_math.sh` in the `cMath` directory was created and used to execute the `RunAllSuite` class, which captured power and energy measurements using the JoularJX and managed the resulting files.

The script sets the Java classpath for using the *JUnit* and the `cpsuite-1.2.6` jars, and performs the `RunAllSuite` java program in a loop for 30 iterations. And, the script is based on the `jx_script.sh` script (see Appendix A.1). So, the steps that it executes are the same than the previous experiment.

4.2.1 Experimental Results and Analysis

In this section, we analyze the energy and power consumption data of the *cMath* project. Figures 4.5 and 4.6 depict the Shapiro-Wilk test results for energy and power data distributions. The test statistics obtained were 0.907 and 0.864 with *p-values* of 0.013 and 0.001, respectively. Given these *p-values* are below the 0.05 significance level, we can reject the null hypothesis and conclude that the data for both measurements, *i.e.,* energy and power consumption, is not normally distributed.

```
---- Shapiro-Wilk Test Results ----
Test statistic: 0.9073889255523682
P-value: 0.012794420123l0028
The data does not follow a normal
distribution(reject null hypothesis)
```

Figure 4.5: Shapiro-Wilk test results for Energy(`cMath` project)

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

Figure 4.6: Shapiro-Wilk test results for Power(`cMath` project)

Moreover, Figures 4.7 and 4.8 shows the box-plot for the data distribution of energy and power consumption of process IDs (PID) from the execution of the Java application. The energy consumption varies between 1200 and 1320 joules with an average range of 1240 to 1320 joules, showing a noticeable spike between PIDs 54000 and 54500. Meanwhile, power consumption ranges from 10 to 80 Watts, averaging between 5 and 40 wards, with two notable peaks near 80 wards between PIDs 53500 and 54000.

[2]`cMath` was taken from the bench-marking used in [Sahin et al., 2014] https://bitbucket.org/udse/refactoring-study/src/master/

[3]http://www.java2s.com/Code/Jar/c/Downloadcpsuite126jar.htm

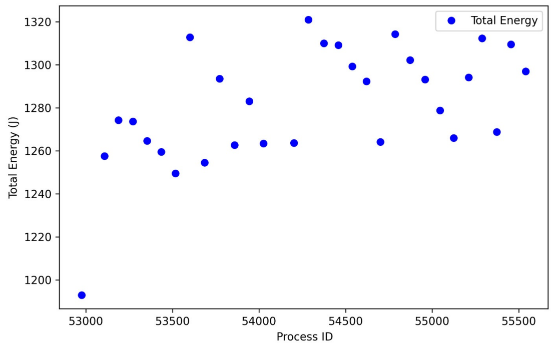


Figure 4.7: Graph of total energy consumption(`cMath` project)

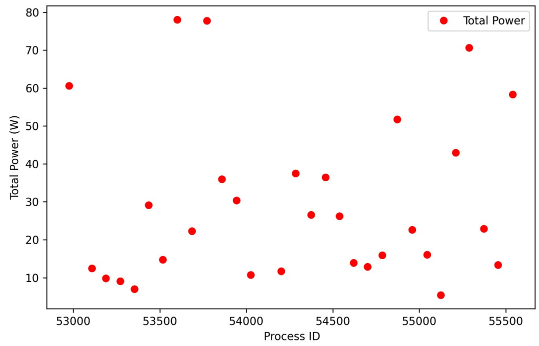


Figure 4.8: Graph of total power consumption(`cMath` project)

In the analysis of the energy consumption for the *cMath* project, as depicted in box-plot figure 4.9, the proximity of the mean and median lines suggests a symmetric distribution, though the mean is slightly higher, hinting at some high consumption values. This box-plot provides a comprehensive view of the project's typical energy usage, essential for its effective management.

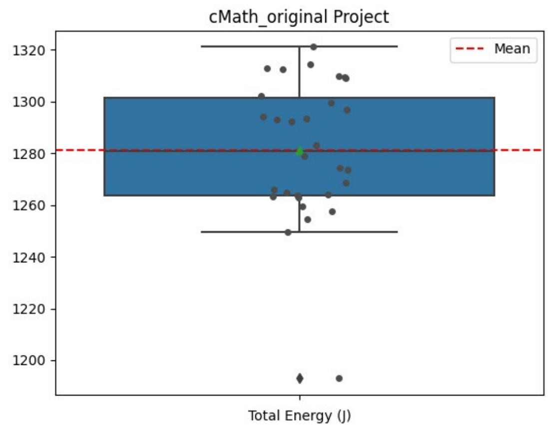


Figure 4.9: Box-plot of total energy consumption(`cMath` project)

Chapter 5

Proposal: Genetic Improvement toward Energy Efficiency

In this chapter, we describe our proposal of using genetic improvement, *i.e. the Gin toolbox*, for software energy efficiency. To do that, we investigate in which extent three fitness functions can reduce energy consumption. Then, Next steps for integrating code refactoring to the gin toolbox are also provided in Section 6.2 of Chapter 6.

5.1 Tactics for reducing energy consumption

When using the Gin tool to improve a method within a program, we need to determine the appropriate fitness function. This function will guide the optimization process, ultimately yielding an improved version of the selected method for the program. Based on our research question **RQ2.1**, we selected two types of fitness functions. These are:

1. Single criteria fitness function
   - **Tactic 1:** Minimize Execution Time
     - **Objective:** Reduce the execution time of the program. Using this tactic, the 'gin' tool will generate an optimal patch that corresponds to the shortest execution time. This approach ensures that the code associated with the optimal patch not only compiles successfully but also enhances the program's execution time performance.
   - **Tactic 2:** Minimize Memory Consumption
     - **Objective:** Decrease the memory usage of the program. With this tactic, the 'gin' tool will produce an optimal patch leading to reduced memory consumption. This method ensures that the code linked with the optimal patch compiles successfully and improves the program's memory consumption performance. Gin's default `LocalSearch` Java class was primarily designed for execution time minimization. Given our specific objective of memory consumption minimization, we made modifications to the `LocalSearch` class.(see Appendix A.3 for more details of updated version of the `LocalSearch` class)
2. Multi criteria fitness function
   - **Tactic 3**: Minimising Execution Time and minimising Memory Consumption.

  - **Objective:** Reduce both the execution time and the memory usage of the program concurrently. Using this tactic, the 'gin' tool will generate an optimal patch that strikes a balance between minimizing execution time and memory consumption. This strategy ensures that the code associated with the optimal patch compiles successfully and offers a harmonized improvement in both the program's execution time and memory consumption performance. We adapted Gin's default `LocalSearch` Java class to minimize both execution time and memory consumption together (see Appendix A.4 for more details of updated version of the `LocalSearch` class). To achieve this, a unique scoring system was devised where both the execution time and memory usage for each patch are normalized and summed. This aggregated score acts as the objective function, with the goal being its minimization. Consequently, a lower score indicates better performance in terms of time and memory. Throughout its iterations, the `LocalSearch` class tests various patches on the source code, selecting and retaining the one that produces the lowest score, thus ensuring efficient code performance.

5.2 Experimental Study Design

As mentioned, the evaluations using Gin were based on single-criteria fitness functions such as *minimizing execution time* and *minimizing memory consumption*, as well as a multi-criteria fitness function related to *minimizing execution time and memory consumption* at the same time.

We selected three programs for our experiments: *Triangle*, *Greatest Common Divisor (GCD)*, and *Rectangle*.

After the three programs were optimized with Gin, we collect data of energy consumption for the original and optimized versions of these programs by using *JoularJX*. Then we compare data gathered from the optimised and original versions of java programs. To determine any significant difference between the data, we used the `Wilcoxon` statistical Test because the data distribution is non-normal.

From these results, we will be able to address **RQ2.1**, which will help us determine whether the optimized version of the program, using the gin tool, has a significant impact on the reduction of energy consumption.

5.2.1 Experimental Procedure to use Gin

To run our experiments, we sequentially executed 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 java class file (*e.g.,* TriangleTest.java) :** 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 the local search on a simple example:**

   ```
   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.

5.3 Experimental Results using Gin

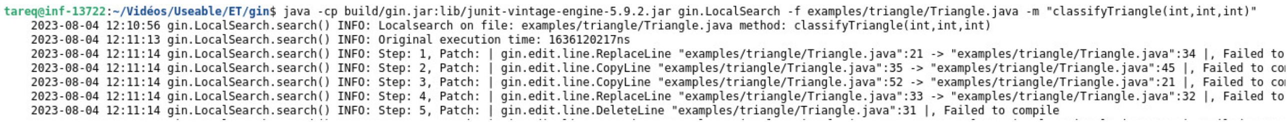
5.3.1 Triangle Program

The *Triangle* code classifies triangles, given their side lengths, into one of four categories: invalid, equilateral, isosceles, or scalene. By sorting the input sides and analyzing their relationships, the program identifies and returns the type of triangle.

Specifically, our goal was to enhance the performance of the `classifyTriangle` method within the *Triangle* program (see Appendix A.5). We targeted the three fitness criteria mentioned in Section 5.1 to optimize the triangle code using Gin. Then optimized and original versions of the *Triangle* code are compare in terms of their energy consumption.

Single criteria Fitness Function: Tactic 1(Minimizing execution time)

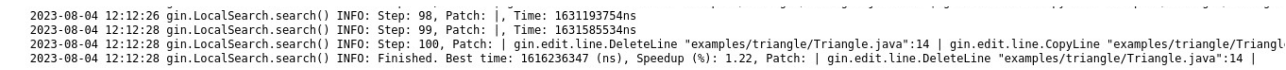
For the *triangle* java code, we minimize its execution time by using the Gin toolbox. Figure 5.1 shows the execution of Gin on the *triangle* code. As seen, it provides information about each step of the local search execution. It started with a summary of the file and method being analyzed, followed by reporting the original execution time of the method. As Figure 5.1 shows, the execution time for the original version of the *triangle* java code was 1636120217ns.



```
tareq@inf-13722:~/Vidéos/Useable/ET/gin$ 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)"
    2023-08-04 12:10:56 gin.LocalSearch.search() INFO: Localsearch on file: examples/triangle/Triangle.java method: classifyTriangle(int,int,int)
    2023-08-04 12:11:13 gin.LocalSearch.search() INFO: Original execution time: 1636120217ns
    2023-08-04 12:11:14 gin.LocalSearch.search() INFO: Step: 1, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":21 -> "examples/triangle/Triangle.java":34 |, Failed to
    2023-08-04 12:11:14 gin.LocalSearch.search() INFO: Step: 2, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":35 -> "examples/triangle/Triangle.java":45 |, Failed to co
    2023-08-04 12:11:14 gin.LocalSearch.search() INFO: Step: 3, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":52 -> "examples/triangle/Triangle.java":21 |, Failed to co
    2023-08-04 12:11:14 gin.LocalSearch.search() INFO: Step: 4, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":33 -> "examples/triangle/Triangle.java":32 |, Failed to
    2023-08-04 12:11:14 gin.LocalSearch.search() INFO: Step: 5, Patch: | gin.edit.line.DeleteLine "examples/triangle/Triangle.java":31 |, Failed to compile
```

Figure 5.1: Command line output: Optimized Triangle Java program with Gin tool.

For each step, the output indicated the patch being applied to the code. If a patch fails to compile, it means that the modified code could not be compiled successfully and this patch is discarded.If the patched code is complied and passes all provided tests, the execution time is calculated and showed if it better to previous version of code.



```
    2023-08-04 12:12:26 gin.LocalSearch.search() INFO: Step: 98, Patch: |, Time: 1631193754ns
    2023-08-04 12:12:28 gin.LocalSearch.search() INFO: Step: 99, Patch: |, Time: 1631585534ns
    2023-08-04 12:12:28 gin.LocalSearch.search() INFO: Step: 100, Patch: | gin.edit.line.DeleteLine "examples/triangle/Triangle.java":14 | gin.edit.line.CopyLine "examples/triangle/Triangl
    2023-08-04 12:12:28 gin.LocalSearch.search() INFO: Finished. Best time: 1616236347 (ns), Speedup (%): 1.22, Patch: | gin.edit.line.DeleteLine "examples/triangle/Triangle.java":14 |
```

Figure 5.2: Command line output: Optimized Triangle Java program with Gin tool.

The local search algorithm continued for a specified number of steps (condition for stopping the Gin process), which was set to 100 by default (see Figure 5.2). Throughout the process, different patches were generated and evaluated, aiming to find an optimized version of them.By examining the output, we could identify the patches that resulted in successfully compiled code and improved execution time. At the end of the process, a brief summary was provided to give an overview of the optimization results. Best time(Lowest execution time): 1616236347 (ns), Speedup (%): 1.22.

Figure 5.3 shows that the optimized code was generated. 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 green color represents the original program file of the Triangle program, which is the code in its initial state without any optimization.

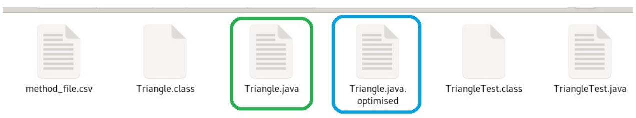


Figure 5.3: Optimised program file of Triangle

Single criteria Fitness Function: Tactic 2(Minimizing memory consumption)

For the *triangle* java code, we minimize its memory consumption by using the Gin toolbox. Figure 5.4 shows the execution of Gin on the *triangle* code. As seen, it provides information about each step of the local search execution. It started with a summary of the file and method being analyzed, followed by reporting the original memory consumption of the method. As Figure 5.4 shows, the memory consumption for the original version of the *triangle* java code was 28 Mbytes.

```
gin.LocalSearch.search() INFO: Localsearch on file: examples/triangle/Triangle.java method: classifyTriangle(int,int,int)
gin.LocalSearch.search() INFO: Original memory consumption: 28 Mbytes
gin.LocalSearch.search() INFO: Step: 1, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":21 -> "examples/triangle/Triangle.java":34 |, Failed to pass all tests
gin.LocalSearch.search() INFO: Step: 2, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":35 -> "examples/triangle/Triangle.java":45 |, Failed to compile
gin.LocalSearch.search() INFO: Step: 3, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":52 -> "examples/triangle/Triangle.java":21 |, Failed to compile
```

Figure 5.4: Command line output: Memory Consumption Optimized Triangle Java program with Gin tool.

For each step, the output indicated the patch being applied to the code. If a patch fails to compile, it means that the modified code could not be compiled successfully and this patch is discarded. If the patched code is complied and passes all provided tests, the memory consumption is calculated and showed if it better to previous version of code.

The local search algorithm continued for a specified number of steps (condition for stopping the Gin process), which was set to 100 by default (see Figure 5.5). Throughout the process, different patches were generated and evaluated, aiming to find an optimized version of them. By examining the output, we could identify the patches that resulted in successfully compiled code and improved memory consumption. At the end of the process, a brief summary was provided to give an overview of the optimization results. Best memory consumption: 22 Mbytes, Memory Consumption Reduction: 21.43%.

```
gin.LocalSearch.search() INFO: Step: 98, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":33 -> "examples/triangle/Triangle.java":41 |, Memory: 36 Mbytes
gin.LocalSearch.search() INFO: Step: 99, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":26 -> "examples/triangle/Triangle.java":47 |, Failed to compile
gin.LocalSearch.search() INFO: Step: 100, Patch: | gin.edit.line.SwapLine "examples/triangle/Triangle.java":31 <-> "examples/triangle/Triangle.java":37 |, Failed to pass all tests
gin.LocalSearch.search() INFO: Finished. Best memory consumption: 22 Mbytes, Memory reduction: 21.43%, Patch: |
```

Figure 5.5: Command line output: Memory Consumption Optimized Triangle Java program with Gin tool.

Finally, in the same directory where the original Triangle program is, the optimized code of Triangle program for memory consumption was generated based on the best patch that consumed the lowest memory.

Multi criteria Fitness Function: Tactic 3(Minimizing execution time and memory consumption)

For the *triangle* java code, we minimize its execution time and memory consumption together by using the Gin toolbox. Figure 5.6 shows the execution of Gin on the *triangle* code. As seen, it provides information about each step of the local search execution. It started with a summary of the file and method being analyzed, followed by reporting the original execution time and memory consumption of the method. As Figure 5.6 shows, the execution time and memory consumption for the original version of the *triangle* java code was 1773092415 ns and 30 Mbytes.

```
gin.LocalSearch.executeSearch() INFO: Localsearch on file: examples/triangle/Triangle.java method: classifyTriangle(int,int,int)
gin.LocalSearch.executeSearch() INFO: Original execution time: 1773092415 ns
gin.LocalSearch.executeSearch() INFO: Original memory consumption: 30 Mbytes
gin.LocalSearch.executeSearch() INFO: Step: 1, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":21 -> "examples/triangle/Triangle.java":34 |, Failed to pass all tests
gin.LocalSearch.executeSearch() INFO: Step: 2, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":35 -> "examples/triangle/Triangle.java":45 |, Failed to compile
gin.LocalSearch.executeSearch() INFO: Step: 3, Patch: | gin.edit.line.CopyLine "examples/triangle/Triangle.java":52 -> "examples/triangle/Triangle.java":21 |, Failed to compile
gin.LocalSearch.executeSearch() INFO: Step: 4, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":33 -> "examples/triangle/Triangle.java":32 |, Failed to compile
gin.LocalSearch.executeSearch() INFO: Step: 5, Patch: | gin.edit.line.DeleteLine "examples/triangle/Triangle.java":31 |, Failed to compile
```

Figure 5.6: Command line output: Execution time and Memory Consumption Optimized Triangle Java program with Gin tool.

For each step, the output indicated the patch being applied to the code. If a patch fails to compile, it means that the modified code could not be compiled successfully and this patch is discarded.If the patched code is complied and passes all provided tests, the execution time and memory consumption are calculated and showed if it better to previous version of code.

The local search algorithm continued for a specified number of steps (condition for stopping the Gin process), which was set to 100 by default (see Figure 5.7). Throughout the process, different patches were generated and evaluated, aiming to find an optimized version of them. By examining the output, we could identify the patches that resulted in successfully compiled code and improved execution time and memory consumption together. At the end of the process, a brief summary was provided to give an overview of the optimization results. Best time(Lowest execution time): 1751420953 (ns), Speedup (%): 1.22, Best memory consumption: 26 Mbytes, Memory reduction: 13.33%.

```
gin.LocalSearch.executeSearch() INFO: Step: 99, Patch: | gin.edit.line.ReplaceLine "examples/triangle/Triangle.java":26 -> "examples/triangle/Triangle.java":47 |, Failed to compile
gin.LocalSearch.executeSearch() INFO: Step: 100, Patch: | gin.edit.line.SwapLine "examples/triangle/Triangle.java":31 <-> "examples/triangle/Triangle.java":37 |, Failed to pass all tests

gin.LocalSearch.executeSearch() INFO: Finished. Best time: 1751420953 (ns), Speedup (%): 1.22, Best memory consumption: 26 Mbytes, Memory reduction: 13.33%, Patch: |
```

Figure 5.7: Command line output: Execution time and Memory Consumption Optimized Triangle Java program with Gin tool.

Finally, in the same directory where the original Triangle program is, the optimized code of Triangle program for execution time and memory consumption together was generated. This optimization was based on the best patch, which yielded the lowest memory consumption and lowest execution time.

5.3.2 Greatest Common Divisor(GCD) and Rectangle Program

The *GCD* Program computes the greatest common divisor (GCD) for three input integers. The fundamental purpose of this program is to determine the largest integer that can evenly divide all three numbers without leaving a remainder. To achieve this, the program utilizes the `findGCD` method.

The *Rectangle* program categorizes a four-sided polygon based on its side lengths into one of three classifications: square, rectangle, or invalid shape. By evaluating the dimensions of the given sides, the program discerns the type of quadrilateral. If all four sides are the same length, it is identified as a square. If the opposite sides are equal, it is recognized as a rectangle. Conversely, if any side is non-positive or if the lengths don't fit the criteria for a square or rectangle, the shape is labeled as invalid. To achieve this, the program utilizes the `classifyRectangle`.

Specifically, our goal was to enhance the performance of the `findGCD` and `classifyRectangle` method within the *GCD* and *Rectangle* program (see Appendix A.6 and A.7). We targeted the three fitness criteria mentioned in Section 5.1 to optimize the triangle code using Gin. Then optimized and original versions of the *GCD* and *Rectangle* code are compare in terms of their energy consumption.

To optimize the two programs(*GCD* and *Rectangle*), using the Gin tool, we followed the same procedures as described for the *Rectangle* program in Section 5.3.1.

5.4 Experimental Results using JoularJX

In the previous section, we obtained the optimization of selected programs (*i.e., Triangle*, *GCD*, *Rectangle*) using the Gin toolbox. In this section, we aim to answer our research question **RQ:2.1** (*i.e., Does the improvement of execution time and memory consumption reduce energy consumption?*).

To do that, we first collected energy consumption measurements of the executions of the two versions, *i.e.,* original and optimised versions, of the studied programs. This was done by using *JoularJX*. For this, we reused the same experimental procedures introduced in our preliminary study (*i.e.,* Section 4.1 that details the monitoring of the energy consumed by a single program), and made modifications to the script (the `jx_script.sh` is presented in Appendix A.1) to adapt it to the studied programs. These procedures were followed for each original and optimized version of the selected programs to monitor their energy consumption. We then use the `Wilcoxon` statistical test to determine any significance difference of the consumed energy of the versions of the studied programs.

For the energy consumption monitoring, we executed 30 times each version of each studied program using the previous script, it means that we isolated each monitoring. For instance, for evaluating the effect of the **Tactic 1**, *i.e.,* minimizing execution time, we first monitored the energy consumption of the original version of the *Triangle* program. Then, we monitored the energy consumption of the optimized version for *Triangle* program. Finally, we compared collected data to evaluate if there is any significance difference between them. We repeat the same process for the other tactic, *i.e.,* **Tactic 2** - *minimizing memory consumption* and **Tactic 3** - *minimizing execution time and memory consumption*, and for the others programs, *i.e.,* the *GCD* and the *Rectangle* programs.

Table 5.1 summarises the results obtained after performing our experimental study. As we can observe, we introduce per each version of the studied programs (see Column 1) the values obtained for each tactics applied. These obtained values correspond to the fitness values that were optimised and the corresponding energy consumption. Notice that we calculated the difference between energy consumed by the two versions of the programs (see the energy consumption values in Columns 3, 5 and 7 of the optimised versions of programs in Rows 2, 4 and 6). Finally, we identify the best tactic to reduce energy consumption per program (see Column 8).

Version of Program	**Tactic 1: min ET**		**Tactic 2: min MC**		**Tactic 3: min ET + MC**		**Best Tactic for Reducing EC**
	ET (s)	**EC (J)**	**MC (Mbytes)**	**EC (J)**	**ET + MC**	**EC (J)**	
Original Triangle	1.636 s	Mean:5.834 [Confidence Interval:95% (5.391,6.276)] SD:1.185 Relative SD:0.203	28 Mbytes	Mean:7.241 [Confidence Interval:95% (6.968,7.514)] SD:0.731 Relative SD:0.100	1.773 s 30 Mbytes	Mean:8.282 [Confidence Interval:95% (7.775,8.789)] SD: 1.358 Relative SD:0.164	**Tactic 3**: Execution time, Memory Consumption
Optimised Triangle	1.616 s Speedup: 1.22%	Mean:5.023 [Confidence Interval:95% (4.579,5.467)] SD:1.189 Relative SD:0.237 **Energy consumption reduction:** ∼ 13.9%	22 Mbytes Memory reduction: 21.43%	Mean:6.051 [Confidence Interval:95% (5.738,6.365)] SD: 0.839 Relative SD:0.138 **Energy consumption reduction:** ∼ 16.43%	1.751 s Speedup: 1.22%. 26 Mbytes Memory reduction: 13.33%	Mean:5.955 [Confidence Interval:95% (5.697,6.212)] SD: 0.689 Relative SD:0.116 **Energy consumption reduction:** ∼ 28.11%	**Energy consumption reduction:** ∼ 28.11%
Original GCD	2.813 s	Mean:27.766 [Confidence Interval:95% (26.889,28.643)] SD:2.349 Relative SD:0.085	180 Mbytes	Mean:26.353 [Confidence Interval:95% (25.537,27.169)] SD:2.186 Relative SD:0.083	2.892 s 158 Mbytes	Mean:26.747 [Confidence Interval:95% (25.996,27.498)] SD:2.011 Relative SD:0.075	**Tactic 3**: Execution time, Memory Consumption
Optimised GCD	2.462 s Speedup: 12.48%	Mean:13.551 [Confidence Interval:95% (13.103,13.999)] SD: 1.199 Relative SD:0.088 **Energy consumption reduction:** ∼ 51.16%	28 Mbytes Memory reduction: 84.44%	Mean:12.848 [Confidence Interval:95% (12.406,13.290)] SD:1.184 Relative SD:0.092 **Energy consumption reduction:** ∼ 51.26%	2.631 s Speedup: 9.00% 23 Mbytes Memory reduction: 85.44%	Mean:12.964 [Confidence Interval:95% (12.554,13.375)] SD:1.099 Relative SD:0.084 **Energy consumption reduction:** ∼ 51.53%	**Energy consumption reduction:** ∼ 51.53%

Original Rectangle	2.629 s	Mean:6.587 [Confidence Interval:95% (6.027,7.147)] SD:1.499 Relative SD:0.227	26 Mbytes	Mean:6.431 [Confidence Interval:95% (5.781,7.081)] SD:1.741 Relative SD:0.270	2.707 s 31 Mbytes	Mean:7.411 [Confidence Interval:95% (6.926,7.895)] SD:1.297 Relative SD:0.175	**Tactic 1:** Execution Time
Optimised Rectangle	1.213 s Speedup: 53.86%	Mean:5.217 [Confidence Interval:95% (4.908,5.527)] SD:0.828 Relative SD:0.158 **Energy consumption reduction:** $\sim$ 20.78%	14 Mbytes Memory reduction: 46.15%	Mean:5.299 [Confidence Interval:95% (4.705,5.895)] SD:1.594 Relative SD:0.300 **Energy consumption reduction:** $\sim$ 17.6%	1.286 s Speedup: 52.59% 20 Mbytes Memory reduction: 35.48%	Mean:6.340 [Confidence Interval:95% (5.815,6.866)] SD:1.407 Relative SD:0.221 **Energy consumption reduction:** $\sim$ 14.45%	**Energy consumption reduction:** $\sim$ 20.78%

Table 5.1: Comparison of the energy consumed by the original vs. the optimized versions of the studied programs, *where: ET=execution time in seconds, MC=memory consumption in Megabytes, EC=energy consumption in Joules*

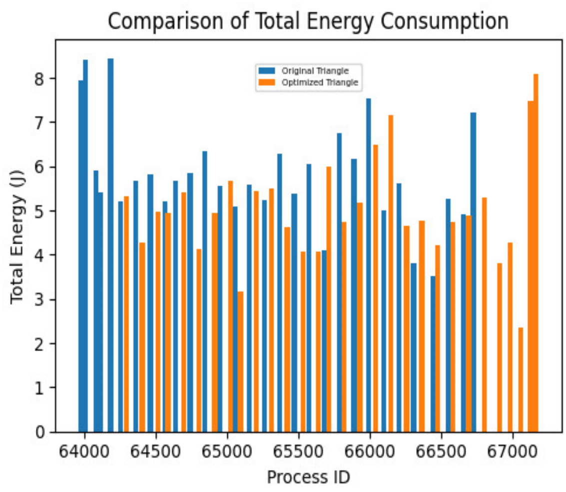
In the following sections we provide the analysis of our results per program and then we discuss the results to get some general conclusions of this study.

5.4.1 Energy Consumption comparison for the original vs. the optimized versions of the *Triangle* Program

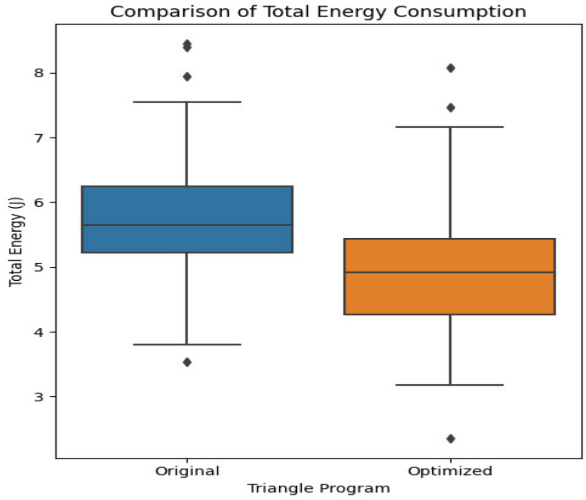
As we can observe in Table 5.1, the Gin toolbox was able to optimised the *Triangle* program by using **Tactic 1 - minimizing the execution time**. The obtained execution time was 1.636 seconds for the original version and 1.616 seconds for the optimised version, thus reducing 1.22% of execution time (see Column 2 of Rows 1, 2) by applying Tactic 1. Regarding energy consumption, we obtained 5.834 joules for the original version and we obtained 5.023 joules for the optimised version. Thus, the energy consumption was reduced in 13.9% by the optimised program (see Column 3 of Rows 1, 2) by applying Tactic 1. We can also observe this reduction of energy consumption in Figures 5.8(a) and 5.8(b) that shows the histogram and the box-plot of the data collected from 30 executions. Notice that the reported energy consumption are the mean values of energy consumption measurements collected from 30 executions. All of these mean values have confident error margin, *i.e.,* the relative standard deviation or coefficient of variance was lower than 5%.

Moreover, the Gin toolbox was able to optimised the *Triangle* program by using **Tactic 2 - minimizing the memory consumption**. The obtained memory consumption was 28 megabytes for the original version and 22 megabytes for the optimised version, thus reducing 21.43% of memory consumption (see Column 4 of Rows 1, 2) by applying Tactic 2. Regarding energy consumption, we obtained 7.241 joules for the original version and we obtained 6.051 joules for the optimised version. Thus, the energy consumption was reduced in 16.43% by the optimised program (see Column 5 of Rows 1, 2) by applying Tactic 2. We can also observe this reduction of energy consumption in Figures 5.8(c) and 5.8(d) that shows the histogram and the box-plot of the data collected from 30 executions.

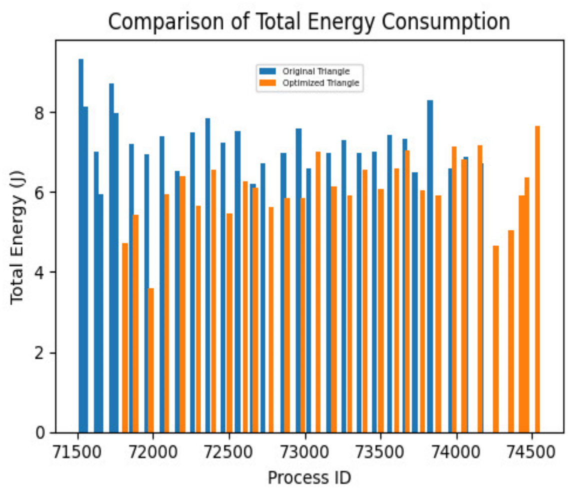
Finally, the Gin toolbox was able to optimised the *Triangle* program by using **Tactic 3 - minimizing execution time and memory consumption**. The obtained fitness function was 1.773 seconds and 30 megabytes for the original version and 1.751 seconds and 26 megabytes for the optimised



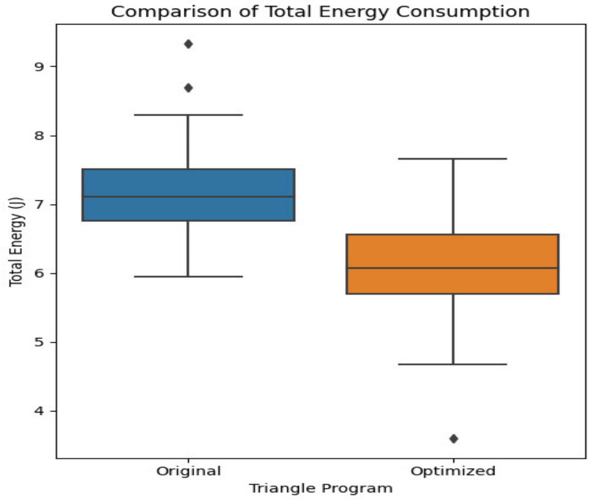
(a) **Tactic 1** - Original vs. Optimized Triangle Histogram



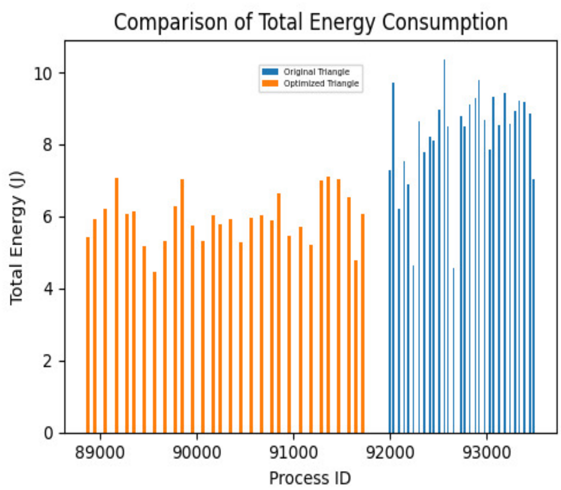
(b) **Tactic 1** - Original vs. Optimized Triangle Boxplot



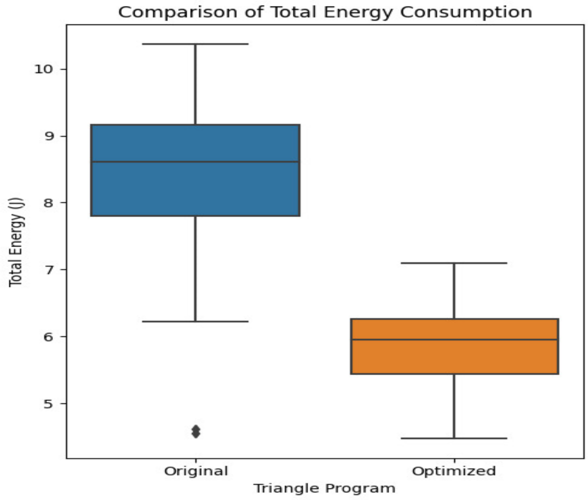
(c) **Tactic 2** - Original vs. Optimized Triangle Histogram



(d) **Tactic 2** - Original vs. Optimized Triangle Boxplot



(e) **Tactic 3** - Original vs. Optimized Triangle Histogram



(f) **Tactic 3** - Original vs. Optimized Triangle Boxplot

Figure 5.8: Energy consumption comparison for the original vs. optimized versions of the *Triangle* program by applying the three studied tactics

version, thus reducing 1.22% of execution time and 13.33% of memory consumption (see Column 6 of Rows 1, 2) by applying Tactic 3. Regarding energy consumption, we obtained 8.282 joules for the original version and we obtained 5.955 joules for the optimised version. Thus, the energy consumption was reduced in 28.11% by the optimised program (see Column 7 of Rows 1, 2) by applying Tactic 3. We can also observe this reduction of energy consumption in Figures 5.8(e) and 5.8(f) that shows the histogram and the box-plot of the data collected from 30 executions.

We observe that for the *Triangle* program the best tactic for reducing energy consumption was **Tactic 3 - minimizing execution time and memory consumption**. This tactic was able to reduce in 28.11% the energy consumed by the *Triangle* program (see Column 8 of Rows 1, 2).

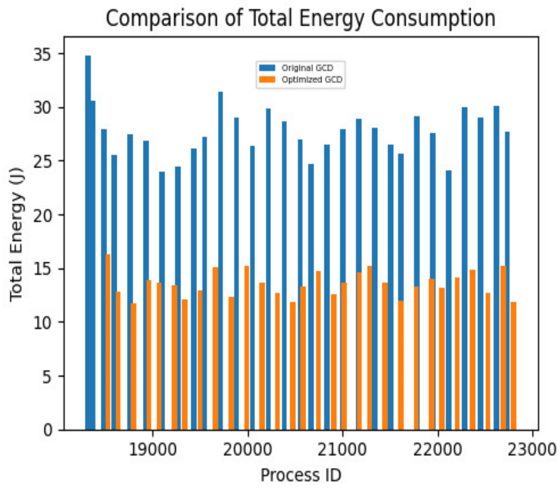
Notice that when a multi-criteria fitness function is used to optimise a code, each fitness value is optimised in a less percentage than optimising the code by using a single fitness function. For instance, while the percentage of the reduced execution time for Tactic 1 and Tactic 3 are equivalent, the percentage for the consumed memory was penalised from 21.43% to 13.33%. Instead of this penalisation, Tactic 3 is the most effective tactic to reduce energy consumption for the *Triangle* program. It could be due to the energy consumed by programs does not only depends on execution time but also depends on the use of CPU. Therefore, if we optimise the memory consumption it could help to reduce the use of CPU, thus reducing energy consumption.

5.4.2 Energy Consumption comparison for the original vs. the optimized versions of the *Greatest Common Divisor(GCD)* Program

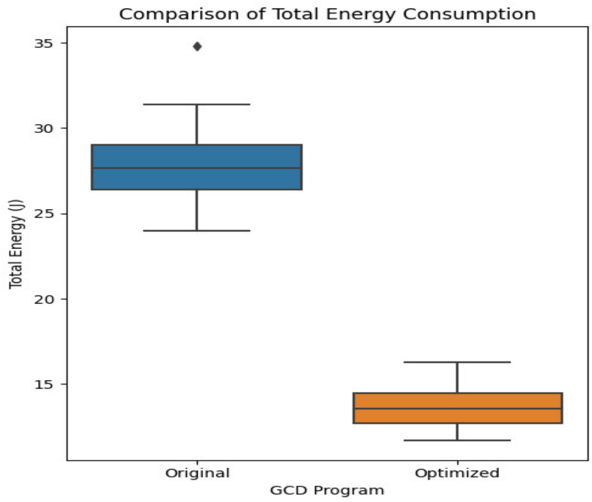
As we can observe in Table 5.1, the Gin toolbox was able to optimised the *Greatest Common Divisor(GCD)* program by using **Tactic 1 - minimizing the execution time**. The obtained execution time was 2.813 seconds for the original version and 2.462 seconds for the optimised version, thus reducing 12.48% of execution time (see Column 2 of Rows 3, 4) by applying Tactic 1. Regarding energy consumption, we obtained 27.766 joules for the original version and we obtained 13.551 joules for the optimised version. Thus, the energy consumption was reduced in 51.16% by the optimised program (see Column 3 of Rows 3, 4) by applying Tactic 1. We can also observe this reduction of energy consumption in Figures 5.9(a) and 5.9(b) that shows the histogram and the box-plot of the data collected from 30 executions. Notice that the reported energy consumption are the mean values of energy consumption measurements collected from 30 executions. All of these mean values have confident error margin, *i.e.,* the relative standard deviation or coefficient of variance was lower than 5%.

Moreover, the Gin toolbox was able to optimised the *Greatest Common Divisor(GCD)* program by using **Tactic 2 - minimizing the memory consumption**. The obtained memory consumption was 180 megabytes for the original version and 28 megabytes for the optimised version, thus reducing 84.44% of memory consumption (see Column 4 of Rows 3, 4) by applying Tactic 2. Regarding energy consumption, we obtained 26.353 joules for the original version and we obtained 12.848 joules for the optimised version. Thus, the energy consumption was reduced in 51.26% by the optimised program (see Column 5 of Rows 3, 4) by applying Tactic 2. We can also observe this reduction of energy consumption in Figures 5.9(c) and 5.9(d) that shows the histogram and the box-plot of the data collected from 30 executions.

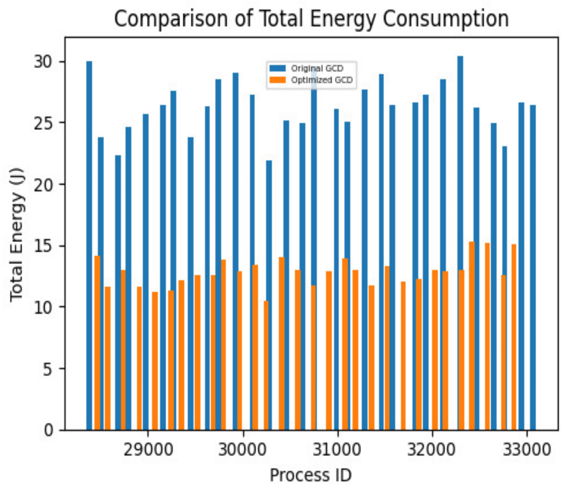
Finally, the Gin toolbox was able to optimised the *Greatest Common Divisor(GCD)* program by using **Tactic 3 - minimizing execution time and memory consumption**. The obtained fitness function was 2.892 seconds and 158 megabytes for the original version and 2.631 seconds and 23 megabytes for the optimised version, thus reducing 9.00% of execution time and 85.44% of memory



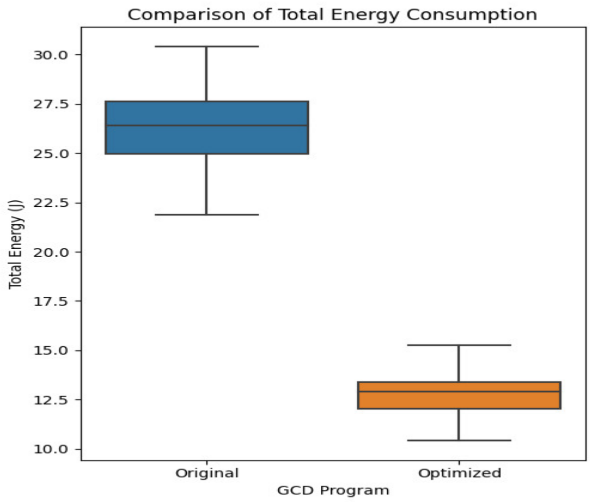
(a) **Tactic 1** - Original vs. Optimized GCD Histogram



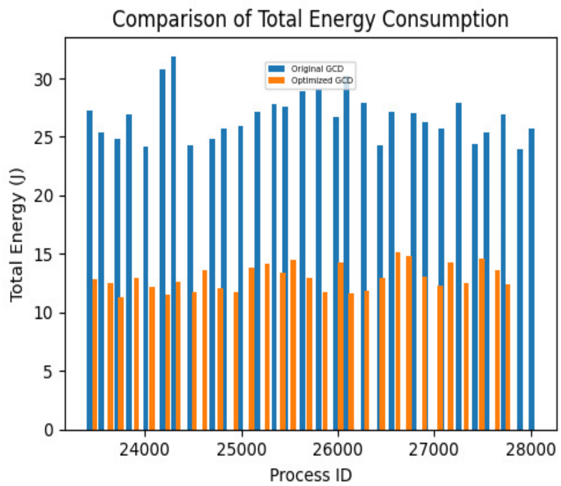
(b) **Tactic 1** - Original vs. Optimized GCD Boxplot



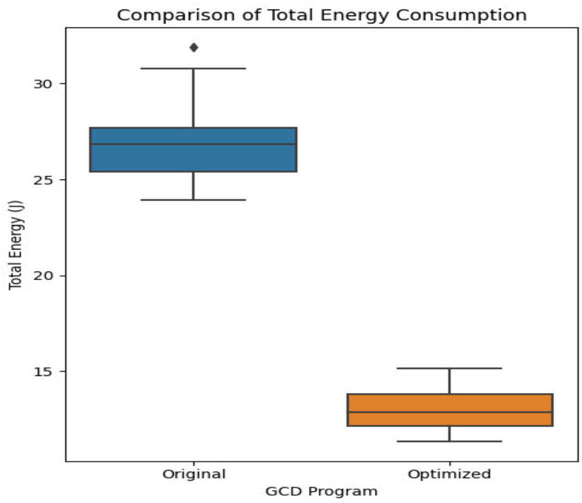
(c) **Tactic 2** - Original vs. Optimized GCD Histogram



(d) **Tactic 2** - Original vs. Optimized GCD Boxplot



(e) **Tactic 3** - Original vs. Optimized GCD Histogram



(f) **Tactic 3** - Original vs. Optimized GCD Boxplot

Figure 5.9: Energy consumption comparison for the original vs. optimized versions of the *Greatest Common Divisor(GCD)* program by applying the three studied tactics

consumption (see Column 6 of Rows 3, 4) by applying Tactic 3. Regarding energy consumption, we obtained 26.747 joules for the original version and we obtained 12.964 joules for the optimised version. Thus, the energy consumption was reduced in 51.53% by the optimised program (see Column 7 of Rows 3, 4) by applying Tactic 3. We can also observe this reduction of energy consumption in Figures 5.9(e) and 5.9(f) that shows the histogram and the box-plot of the data collected from 30 executions.

We observe that for the *Greatest Common Divisor(GCD)* program the best tactic for reducing energy consumption was **Tactic 3 - minimizing execution time and memory consumption**. This tactic was able to reduce in 51.53% the energy consumed by the *Greatest Common Divisor(GCD)* program (see Column 8 of Rows 3, 4).

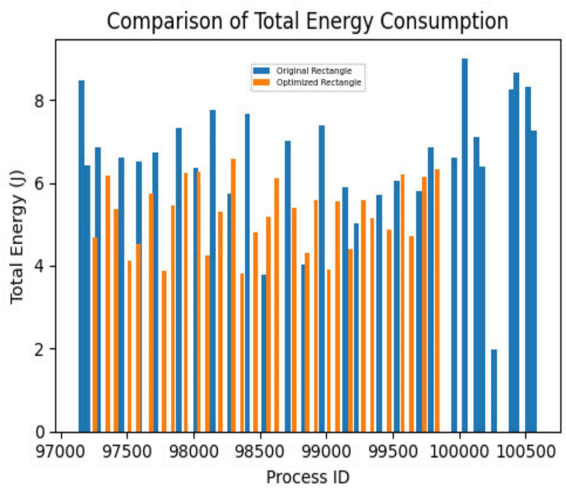
Notice that when a multi-criteria fitness function is used to optimise a code, each fitness value is optimised in a less percentage than optimising the code by using a single fitness function. For instance, the percentage for the execution time was penalised from 12.48% to 9.00% in Tactic 1 to Tactic 3. Instead of this penalisation, Tactic 3 is the most effective tactic to reduce energy consumption for the *Greatest Common Divisor(GCD)* program. It could be due to the energy consumed by programs does not only depends on execution time but also depends on the use of memory. Therefore, since Tactic 3 contains the highest memory reduce percentage it leads to reduction in energy consumption.

5.4.3 Energy Consumption comparison for the original vs. the optimized versions of the *Rectangle* Program

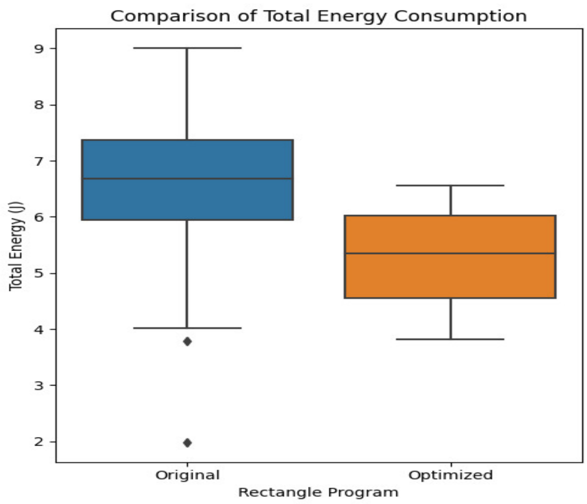
As we can observe in Table 5.1, the Gin toolbox was able to optimised the *Rectangle* program by using **Tactic 1 - minimizing the execution time**. The obtained execution time was 2.629 seconds for the original version and 1.213 seconds for the optimised version, thus reducing 53.86% of execution time (see Column 2 of Rows 5, 6) by applying Tactic 1. Regarding energy consumption, we obtained 6.587 joules for the original version and we obtained 5.217 joules for the optimised version. Thus, the energy consumption was reduced in 20.78% by the optimised program (see Column 3 of Rows 5, 6) by applying Tactic 1. We can also observe this reduction of energy consumption in Figures 5.10(a) and 5.10(b) that shows the histogram and the box-plot of the data collected from 30 executions. Notice that the reported energy consumption are the mean values of energy consumption measurements collected from 30 executions. All of these mean values have confident error margin, *i.e.,* the relative standard deviation or coefficient of variance was lower than 5%.

Moreover, the Gin toolbox was able to optimised the *Rectangle* program by using **Tactic 2 - minimizing the memory consumption**. The obtained memory consumption was 26 megabytes for the original version and 14 megabytes for the optimised version, thus reducing 46.15% of memory consumption (see Column 4 of Rows 5, 6) by applying Tactic 2. Regarding energy consumption, we obtained 6.431 joules for the original version and we obtained 5.299 joules for the optimised version. Thus, the energy consumption was reduced in 17.6% by the optimised program (see Column 5 of Rows 5, 6) by applying Tactic 2. We can also observe this reduction of energy consumption in Figures 5.10(c) and 5.10(d) that shows the histogram and the box-plot of the data collected from 30 executions.

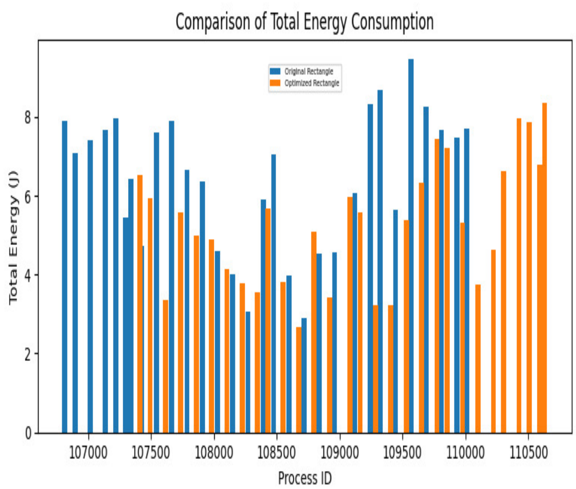
Finally, the Gin toolbox was able to optimised the *Rectangle* program by using **Tactic 3 - minimizing execution time and memory consumption**. The obtained fitness function was 2.707 seconds and 31 megabytes for the original version and 1.286 seconds and 20 megabytes for the optimised version, thus reducing 52.59% of execution time and 35.48% of memory consumption (see Column 6 of Rows 5, 6) by applying Tactic 3. Regarding energy consumption, we obtained 7.411 joules for the original version and we obtained 6.340 joules for the optimised version. Thus, the en-



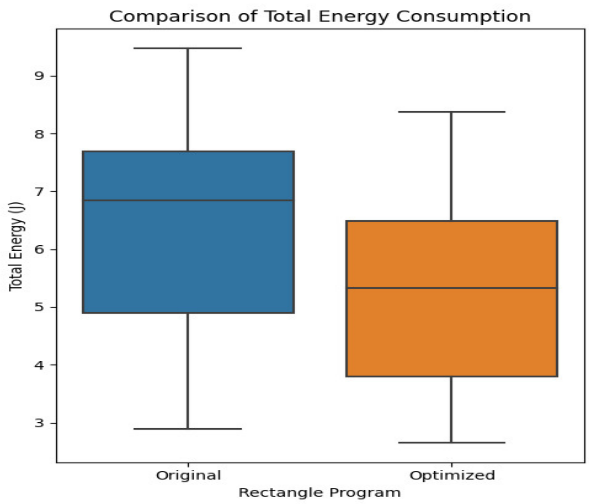
(a) **Tactic 1** - Original vs. Optimized Rectangle Histogram



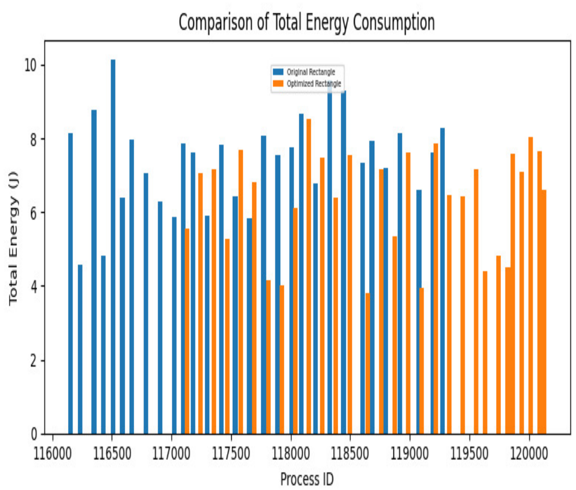
(b) **Tactic 1** - Original vs. Optimized Rectangle Boxplot



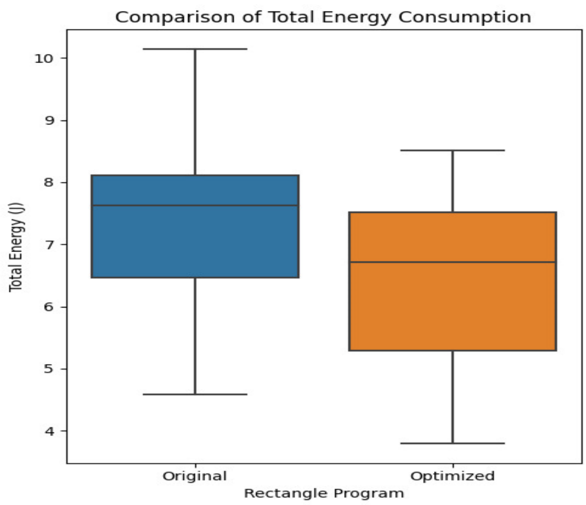
(c) **Tactic 2** - Original vs. Optimized Rectangle Histogram



(d) **Tactic 2** - Original vs. Optimized Rectangle Boxplot



(e) **Tactic 3** - Original vs. Optimized Rectangle Histogram



(f) **Tactic 3** - Original vs. Optimized Rectangle Boxplot

Figure 5.10: Energy consumption comparison for the original vs. optimized versions of the *Rectangle* program by applying the three studied tactics

ergy consumption was reduced in 14.45% by the optimised program (see Column 7 of Rows 5, 6) by applying Tactic 3. We can also observe this reduction of energy consumption in Figures 5.10(e) and 5.10(f) that shows the histogram and the box-plot of the data collected from 30 executions.

We observe that for the *Rectangle* program the best tactic for reducing energy consumption was **Tactic 1 - minimizing the execution time**. This tactic was able to reduce in 20.78% the energy consumed by the *Triangle* program (see Column 8 of Rows 5, 6).

5.4.4 Discussion

In Table 5.1, we observe that the Gin toolbox was able to optimize the *Triangle*, *Greatest Common Divisor (GCD)*, and *Rectangle* programs using **Tactic 1: minimizing the execution time**, **Tactic 2: minimizing memory consumption**, and **Tactic 3: minimizing execution time and memory consumption**, as discussed in Section 5.1. Upon analyzing the table, we observed that the optimized version of each program consistently consumes less energy than its original version. We calculated the energy consumption reduction in percentage for each optimized version and then compared these percentages.

For the *Triangle* program, the energy consumption was reduced by 13.9% with the optimized program when applying **Tactic 1** (see Column 3, Rows 1 and 2). With **Tactic 2**, the reduction was 16.43% (see Column 5, Rows 1 and 2), and with **Tactic 3**, it was 28.11% (see Column 7, Rows 1 and 2). We observe that, for the *Triangle* program, **Tactic 3 - minimizing execution time and memory consumption** was the most effective, achieving a 28.11% reduction in energy consumption (see Column 8, Rows 1 and 2).

For the *Greatest Common Divisor (GCD)* program, the energy consumption was reduced by 51.16% using the optimized program when applying **Tactic 1** (see Column 3, Rows 3 and 4). With **Tactic 2**, the reduction was 51.26% (see Column 5, Rows 3 and 4), and with **Tactic 3**, it was 51.53% (see Column 7, Rows 3 and 4). We observe that **Tactic 3 - minimizing execution time and memory consumption** was the most effective for the *Greatest Common Divisor (GCD)* program, achieving a 51.53% reduction in energy consumption (see Column 8, Rows 3 and 4).

For the *Rectangle* program, the energy consumption was reduced by 20.78% using the optimized program when applying **Tactic 1** (see Column 3, Rows 5 and 6). With **Tactic 2**, the reduction was 17.6% (see Column 5, Rows 5 and 6), and with **Tactic 3**, it was 14.45% (see Column 7, Rows 5 and 6). We observe that, for the *Rectangle* program, **Tactic 1 - minimizing the execution time** was the most effective, achieving a 20.78% reduction in energy consumption (see Column 8, Rows 5 and 6).

To definitively determine which tactic yields the highest energy consumption reduction, further experiments are needed. However, based on our current results, we can conclude that the optimized versions program using the "gin" tool, across all three tactics, consistently consume less energy than their original version program. This evidence supports a positive answer to research question **RQ2.1**: improvements in the execution time and memory consumption of programs do indeed lead to reduced energy consumption.

Chapter 6

Conclusion and Next Steps

6.1 Conclusion

In our internship, the main objective was to investigate tactics for enhancing software energy efficiency. From this objective, we formulated two research questions:

- **RQ1:** Which tactics improve energy efficiency?
- **RQ2:** How can we automate the integration of these tactics to minimize energy consumption?

In response to Research Question 1 (**RQ1**), we delved into various tactics such as Architectural Tactics, Design Patterns, and Code Refactoring. Through a comparative analysis, we discerned the benefits and drawbacks of each tactic. Code refactoring emerged as particularly versatile across various software types. We not only identified these tactics but also detailed insights into their implementation, emphasizing practical methods to embed them in software. Monitoring energy consumption was essential for our research. Among the considered software tools for energy consumption profiling, JoularJX was the most fitting for our focus on Java-based software. This preference stemmed from JoularJX's ability to offer real-time monitoring at the source code level, functioning as a Java agent. To automate the monitoring process using the JoularJX tool, we developed two bash scripts tailored to its updated 2.0 version. Another intern, `Lyne Gabriella NENGUEKO NOUMBISSIE`, utilized these scripts in her research, `Evaluation of energy-efficiency requirements through a systematic test case generation.`

In **RQ2**, we focused on the automation of integrating tactics to reduce energy consumption. This approach utilized genetic improvement, and as part of our research, we introduced a tool called GIN. Under research question **RQ2**, three sub-questions were identified:

- **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?

Analysis of the results from Table 5.1 reveals that the Gin toolbox effectively optimized the *Triangle*, *GCD*, and *Rectangle* programs across three tactics (Section 5.1). Across these programs, the optimized versions consistently used less energy than their originals. For the *Triangle* program, the most energy savings (28.11%) was achieved with **Tactic 3**. For the *GCD* program, energy consumption was most reduced (51.53%) also by **Tactic 3**. However, for the *Rectangle* program, the most reduction (20.78%) was observed with **Tactic 1**. To definitively determine which tactic yields the highest energy consumption reduction, further experiments are needed. However, based on our current results, we can conclude that the optimized versions program using the "gin" tool, across all three tactics, consistently consume less energy than their original version program. This evidence supports a positive answer to research question **RQ2.1**: improvements in the execution time and memory consumption of programs do indeed lead to reduced energy consumption.

A comprehensive literature review was conducted, and it was observed that while various code refactoring techniques could improve software performance, their impact on energy efficiency varied based on the software's specific context. Notably, techniques like 'Convert Local Variable to Field', 'Introduce Parameter Object', and others showed promise in reducing energy consumption. The Genetic Improvement (GIN) tool, which leveraged Genetic Programming for software enhancement, presented an ideal platform for integrating these techniques. The `StatementEdit` class within GIN appeared to be the most fitting for this integration, paving the way for potentially higher software quality and energy efficiency. This study effectively addressed the research question, highlighting the potential synergies between refactoring techniques and the GIN tool. Based on this analysis, research question **RQ2.2** was answered: Code refactoring techniques could indeed be integrated into the Genetic Improvement tool, Gin. For this integration, the `StatementEdit` class in Gin needed to be extended.

Due to time constraints, we were unable to address our research question **RQ2.3**. However, we provide detailed information in the following section, Section 6.2, on how the corresponding experiment can be conducted.

It is very important to make software developers aware of the negative effects of energy consumption. This internship helped me to learn a lot of new things about energy efficiency in the software domain. I am sure that this knowledge will help me a lot in the future. This work made me aware of how important is to reduce energy consumption at the software level and how much energy we can save by using energy-friendly programs or software.

6.2 Next steps

In response to **RQ2.3**, our next steps will involve integrating the selected code refactoring techniques such as `Convert Local Variable to Field`, `Introduce Parameter Object`, and `Move Method` into the identified `StatementEdit` class within Gin. Before this integration, we need to integrate code smell detection technique in the Gin tool. This will allow us for efficient detection and tracing of code smells.

Our first step is to identify if there are any code smells present. Only upon detecting code smells can we then integrate the selected code refactoring techniques. Under this step, we have two sub-steps. The first is to translate the code into Abstract Syntax Tree (AST) format. The second sub-step is to identify the tool that detects code smells from the code in Abstract Syntax Tree (AST) format using specific code smell detection technique. Once identified, the techniques used by the identified code smell detection tool will be integrated into the 'Gin' tool for detecting code smells.

[Liu and Zhang, 2017] presents famous code smeel detection tools: `Checkstyle`, `JDeodorant`. These two tool detect code smells from the code in Abstract Syntax Tree (AST) format using specific code smell detection technique. `Checkstyle` is a well-known static code analysis tool, it can detect 4 code smells Large Class, Long Method, Long Parameter List, and Duplicated Code.`JDeodorant` is an Eclipse plug-in[Tsantalis et al., 2018]. It can automatically detect 4 code smells Feature Envy, God Class, Long Method, and Switch Statement. It can achieve high detection accuracy. In addition, JDeodorant can achieve a good visualization of detection results. But at present it only can detect 4 code smells. `Checkstyle` uses a metric-based approach to detect code smells, where it checks the source code for violations of specified metrics such as cyclomatic complexity, lines of code, and number of parameters[Fontana et al., 2016]. It was not mentioned specifically what approach used in the JDeodorant code smell detection tool.

[dos Reis et al., 2020] presents an up-to-date review of state-of-the-art techniques and tools used for code smell detection and visualization. The study found that the most commonly used approaches for code smell detection are search-based (30.1%), metric-based (24.1%), and symptom-based (19.3%). The study does offer insights into the strengths and limitations of various approaches, which can guide us in deciding the most suitable technique for our specific needs. For instance, search-based approaches are effective in detecting code smells. The effectiveness of a given technique could depend on factors like the specific codebase under analysis, the programming language in use, and the desired accuracy and precision for detecting code smells.

We have selected the search-based code smell detection technique for integration into the Gin tool. Search-based code smell detection is a technique that uses search algorithms to identify instances of code smells in software systems. To integrate search-based code smell detection into GIN, we would need to modify the Gin tool's source code to incorporate the desired search-based technique. This would likely involve implementing the search algorithm and defining a fitness function that evaluates candidate solutions based on their ability to detect code smells. We may also need to modify GIN's existing genetic operators (e.g., mutation and crossover) to work with the new search-based approach.

After integrating code smell detection techniques into the GIN tool, we can proceed to incorporate code refactoring techniques. Once these are integrated, we will conduct experiments using the extended version of the GIN tool to determine whether the optimized version of the program or project results in energy savings in the software domain thereby addressing Research Question **RQ2.3**.

Appendix A

Experimental artifacts

A.1 The jx_script.sh script

```
#!/bin/bash

if [ "$#" -ne 1 ];then
    echo "Un unique argument est attendu : le nom du programme (sans l'extension)" >&2
    exit 1
fi

programme=$1
jar="/opt/joularjx/joularjx-2.0.jar"
mkdir "jx_results"
mkdir "jx_results/energy"
mkdir "jx_results/energy/methods"
mkdir "jx_results/energy/calltrees"
mkdir "jx_results/energy_filtered"
mkdir "jx_results/energy_filtered/methods"
mkdir "jx_results/energy_filtered/calltrees"
mkdir "jx_results/power"
mkdir "jx_results/power/methods"
mkdir "jx_results/power/calltrees"
mkdir "jx_results/power_filtered"
mkdir "jx_results/power_filtered/methods"
mkdir "jx_results/power_filtered/calltrees"

mkdir "mandelbrot_bitmap"

for param in 15000 20000 30000 40000 # Nombre d' lements  dans un tableau
do
    #for i in 1 2  # Nombre d'it rations
    for i in {1..30}  # Nombre d'it rations
    do
  echo "running java  with parameter $param, iteration $i"
  # Nous avons besoin du pid du processus java pour pouvoir correctement g rer les fichiers cr  s par JoularJX
  if [ $i == 1 ];then
      # On sauvegarde les pmb pour la premi re it ration
      java -javaagent:$jar $programme $param > "mandelbrot_bitmap/java_temp.pmb" &
  else
      java -javaagent:$jar $programme $param > /dev/null 2>/dev/null &
  fi
  java_pid=$!
  wait $!

  # Sauvegarde du pmb
  if [ -e "mandelbrot_bitmap/java_temp.pmb" ];then
      tail -n+6 "mandelbrot_bitmap/java_temp.pmb" > "mandelbrot_bitmap/java_$param.pmb"
```

```
    rm -f "mandelbrot_bitmap/java_temp.pmb"
fi

fichier_energy_methods="joularJX-"$java_pid"-all-methods-energy.csv"
fichier_energy_calltrees="joularJX-"$java_pid"-all-call-trees-energy.csv"
fichier_energy_filtered_methods="joularJX-"$java_pid"-filtered-methods-energy.csv"
fichier_energy_filtered_calltrees="joularJX-"$java_pid"-filtered-call-trees-energy.csv"
fichier_power_methods="joularJX-"$java_pid"-all-methods-power.csv"
fichier_power_calltrees="joularJX-"$java_pid"-all-call-trees-power.csv"
fichier_power_filtered_methods="joularJX-"$java_pid"-filtered-methods-power.csv"
fichier_power_filtered_calltrees="joularJX-"$java_pid"-filtered-call-trees-power.csv"

# Tri des fichiers vides
find . -name "$fichier_energy_methods" -type f -print0 | while IFS= read -r -d '' f
do
    if [ -s "$f" ]; then
  # Non-empty file
  mv "$f" "jx_results/energy/methods"
  #echo "Moved $f to jx_results/energy/methods"
    else
  # Empty file
  #echo "Deleting empty file: $f"
  rm -f "$f"
    fi
done
find . -name "$fichier_energy_calltrees" -type f -print0 | while IFS= read -r -d '' f
do
    if [ -s "$f" ]; then
  # Non-empty file
  mv "$f" "jx_results/energy/calltrees"
  #echo "Moved $f to jx_results/energy/calltrees"
    else
  # Empty file
  #echo "Deleting empty file: $f"
  rm -f "$f"
    fi
done
find . -name "$fichier_energy_filtered_methods" -type f -print0 | while IFS= read -r -d '' f
do
    if [ -s "$f" ]; then
  # Non-empty file
  mv "$f" "jx_results/energy_filtered/methods"
  #echo "Moved $f to jx_results/energy_filtered/methods"
    else
  # Empty file
  #echo "Deleting empty file: $f"
  rm -f "$f"
    fi
done
find . -name "$fichier_energy_filtered_calltrees" -type f -print0 | while IFS= read -r -d '' f
do
    if [ -s "$f" ]; then
  # Non-empty file
  mv "$f" "jx_results/energy_filtered/calltrees"
  #echo "Moved $f to jx_results/energy_filtered/calltrees"
    else
  # Empty file
  #echo "Deleting empty file: $f"
  rm -f "$f"
    fi
done
find . -name "$fichier_power_methods" -type f -print0 | while IFS= read -r -d '' f
do
    if [ -s "$f" ]; then
  # Non-empty file
  mv "$f" "jx_results/power/methods"
```

```
    #echo "Moved $f to jx_results/power/methods"
        else
    # Empty file
    #echo "Deleting empty file: $f"
    rm -f "$f"
        fi
  done
  find . -name "$fichier_power_calltrees" -type f -print0 | while IFS= read -r -d '' f
  do
        if [ -s "$f" ]; then
    # Non-empty file
    mv "$f" "jx_results/power/calltrees"
    #echo "Moved $f to jx_results/power/calltrees"
        else
    # Empty file
    #echo "Deleting empty file: $f"
    rm -f "$f"
        fi
  done
  find . -name "$fichier_power_filtered_methods" -type f -print0 | while IFS= read -r -d '' f
  do
        if [ -s "$f" ]; then
    # Non-empty file
    mv "$f" "jx_results/power_filtered/methods"
    #echo "Moved $f to jx_results/power_filtered/methods"
        else
    # Empty file
    #echo "Deleting empty file: $f"
    rm -f "$f"
        fi
  done

  find . -name "$fichier_power_filtered_calltrees" -type f -print0 | while IFS= read -r -d '' f
  do
        if [ -s "$f" ]; then
    # Non-empty file
    mv "$f" "jx_results/power_filtered/calltrees"
    #echo "Moved $f to jx_results/power_filtered/calltrees"
        else
    # Empty file
    #echo "Deleting empty file: $f"
    rm -f "$f"
        fi
  done
    done
done

# used to gather power and engery consumption data using JoularJX and save it t CSV file.
python3 jx_gatherData.py "jx_results/"

#used to analyze the power consumption data at the method level, i.e., it breaks down the data into individual methods and calculates the energy and power consumed by each method.
python3 jx_process_level_methods.py "jx_results/"

# used to generate graphs from the power and engery consumption data saved in the CSV file generated by jx_gatherData.py.
python3 jx_plot.py  "jx_graphs"

# for box plotting the total energy for all the process id
python3 jx_plot_geom_boxplot.py

#used to perform a Shapiro-Wilk test on the energy consumption data to check for normality.
python3 shapiro_wilk_test_energy.py

#used to perform a Shapiro-Wilk test on the power consumption data to check for normality.
python3 shapiro_wilk_test_power.py
```

A.2 The RunAllSuite java code

```
import org.junit.extensions.cpsuite.ClasspathSuite;
import org.junit.extensions.cpsuite.ClasspathSuite.*;
import org.junit.internal.TextListener;
import org.junit.runner.RunWith;
import org.junit.runner.JUnitCore;
import static org.junit.extensions.cpsuite.SuiteType.*;

@RunWith(ClasspathSuite.class)
@SuiteTypes({ JUNIT38_TEST_CLASSES, TEST_CLASSES })
public class RunAllSuite {
        public static void main(String args[]) {
          JUnitCore junit = new JUnitCore();
            junit.addListener(new TextListener(System.out));
            junit.run(RunAllSuite.class);
        }
}
```

A.3 The LocalSearch java code for Tactic 2: Minimize Memory Consumption

```
package gin;

import com.sampullara.cli.Args;
import com.sampullara.cli.Argument;
import gin.edit.Edit;
import gin.edit.Edit.EditType;
import gin.test.InternalTestRunner;
import gin.test.UnitTestResult;
import gin.test.UnitTestResultSet;
import org.apache.commons.io.FilenameUtils;
import org.apache.commons.rng.simple.JDKRandomBridge;
import org.apache.commons.rng.simple.RandomSource;
import org.pmw.tinylog.Logger;

import java.io.File;
import java.io.Serial;
import java.io.Serializable;
import java.util.Collections;
import java.util.List;
import java.util.Random;

/**
 * Simple local search. Takes a source filename and a method signature, optimizes it.
 * Assumes the existence of accompanying Test Class.
 * The class must be in the top level package if classPath not provided.
 */
public class LocalSearch implements Serializable {

    @Serial
    private static final long serialVersionUID = -92020344633720482L;

    private static final int WARMUP_REPS = 10;

    @Argument(alias = "f", description = "Required: Source filename", required = true)
    protected File filename = null;

    @Argument(alias = "m", description = "Required: Method signature including arguments." +
            "For example, \"classifyTriangle(int,int,int)\"", required = true)
    protected String methodSignature = "";

    @Argument(alias = "s", description = "Seed")
```

```
    protected Integer seed = 123;

    @Argument(alias = "n", description = "Number of steps")
    protected Integer numSteps = 100;

    @Argument(alias = "d", description = "Top directory")
    protected File packageDir;

    @Argument(alias = "c", description = "Class name")
    protected String className;

    @Argument(alias = "cp", description = "Classpath")
    protected String classPath;

    @Argument(alias = "t", description = "Test class name")
    protected String testClassName;

    @Argument(alias = "et", description = "Edit type: this can be a member of the EditType enum (LINE,STATEMENT,MATCHED_STATEMENT,MODIFY_STATEMENT); the fully qualified name of a class that extends gin.edit.Edit, or a comma-separated list of both")
    protected String editType = EditType.LINE.toString();

    /**
     * allowed edit types for sampling: parsed from editType
     */
    protected List<Class<? extends Edit>> editTypes;

    @Argument(alias = "ff", description = "Fail fast. "
            + "If set to true, the tests will stop at the first failure and the next patch will be executed. "
            + "You probably don't want to set this to true for Automatic Program Repair.")
    protected Boolean failFast = false;

    protected SourceFile sourceFile;
    protected Random rng;
    InternalTestRunner testRunner;

    // Constructor parses arguments
    LocalSearch(String[] args) {
        Args.parseOrExit(this, args);
        editTypes = Edit.parseEditClassesFromString(editType);

        this.sourceFile = SourceFile.makeSourceFileForEditTypes(editTypes, this.filename.toString(), Collections.singletonList(this.methodSignature));

        this.rng = new JDKRandomBridge(RandomSource.MT, Long.valueOf(seed));
        if (this.packageDir == null) {
            this.packageDir = (this.filename.getParentFile() != null) ? this.filename.getParentFile().getAbsoluteFile() : new File(System.getProperty("user.dir"));
        }
        if (this.classPath == null) {
            this.classPath = this.packageDir.getAbsolutePath();
        }
        if (this.className == null) {
            this.className = FilenameUtils.removeExtension(this.filename.getName());
        }
        if (this.testClassName == null) {
            this.testClassName = this.className + "Test";
        }
        this.testRunner = new InternalTestRunner(className, classPath, testClassName, failFast);
    }

    // Instantiate a class and call search
    public static void main(String[] args) {
        LocalSearch simpleLocalSearch = new LocalSearch(args);
        simpleLocalSearch.search();
```

```java
    }

    // Apply empty patch and return memory consumption
    private long memoryOriginalCode() {
        Patch emptyPatch = new Patch(this.sourceFile);
        UnitTestResultSet resultSet = testRunner.runTests(emptyPatch, WARMUP_REPS);

        if (!resultSet.allTestsSuccessful()) {
            if (!resultSet.getCleanCompile()) {
                Logger.error("Original code failed to compile");
            } else {
                Logger.error("Original code failed to pass unit tests");
                for (UnitTestResult testResult : resultSet.getResults()) {
                    Logger.error(testResult);
                }
            }
            System.exit(0);
        }

        return resultSet.totalMemoryUsage() / WARMUP_REPS;
    }

    // Simple local search
    private void search() {
        Logger.info(String.format("Localsearch on file: %s method: %s", filename, methodSignature));

        // Memory consumption of original code
        long origMemory = memoryOriginalCode();
        Logger.info("Original memory consumption: " + origMemory + " Mbytes");

        // Start with empty patch
        Patch bestPatch = new Patch(this.sourceFile);
        long bestMemory = origMemory;

        for (int step = 1; step <= numSteps; step++) {
            Patch neighbour = neighbour(bestPatch);
            UnitTestResultSet testResultSet = testRunner.runTests(neighbour, 1);

            String msg;

            if (!testResultSet.getValidPatch()) {
                msg = "Patch invalid";
            } else if (!testResultSet.getCleanCompile()) {
                msg = "Failed to compile";
            } else if (!testResultSet.allTestsSuccessful()) {
                msg = "Failed to pass all tests";
            } else if (testResultSet.totalMemoryUsage() >= bestMemory) {
                msg = "Memory: " + testResultSet.totalMemoryUsage() + " Mbytes";
            } else {
                bestPatch = neighbour;
                bestMemory = testResultSet.totalMemoryUsage();
                msg = "New best memory consumption: " + bestMemory + " Mbytes ";
            }

            Logger.info(String.format("Step: %d, Patch: %s, %s ", step, neighbour, msg));
        }

        Logger.info(String.format("Finished. Best memory consumption: %d Mbytes, Memory reduction: %.2f%%, Patch: %s",
                bestMemory,
                100.0 * ((origMemory - bestMemory) / (1.0 * origMemory)),
                bestPatch));

        bestPatch.writePatchedSourceToFile(sourceFile.getRelativePathToWorkingDir() + ".optimised");
    }
```

```

    /**
     * Generate a neighboring patch, either by deleting an edit or adding a new one.
     *
     * @param patch Generate a neighbor of this patch.
     * @return A neighboring patch.
     */
    Patch neighbour(Patch patch) {
        Patch neighbour = patch.clone();

        if (neighbour.size() > 0 && rng.nextFloat() > 0.5) {
            neighbour.remove(rng.nextInt(neighbour.size()));
        } else {
            neighbour.addRandomEditOfClasses(rng, editTypes);
        }

        return neighbour;
    }
}
```

A.4 The LocalSearch java code for Tactic 3: Minimising Execution Time and minimising Memory Consumption together

```
package gin;

import com.sampullara.cli.Args;
import com.sampullara.cli.Argument;
import gin.edit.Edit;
import gin.edit.Edit.EditType;
import gin.test.InternalTestRunner;
import gin.test.UnitTestResult;
import gin.test.UnitTestResultSet;
import org.apache.commons.io.FilenameUtils;
import org.apache.commons.rng.simple.JDKRandomBridge;
import org.apache.commons.rng.simple.RandomSource;
import org.pmw.tinylog.Logger;

import java.io.File;
import java.io.Serial;
import java.io.Serializable;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import java.util.Scanner;

public class LocalSearch implements Serializable {

    @Serial
    private static final long serialVersionUID = -92020344633720482L;

    private static final int WARMUP_REPS = 10;

    @Argument(alias = "f", description = "Required: Source filename", required = true)
    protected File filename = null;

    @Argument(alias = "m", description = "Required: Method signature including arguments." +
            "For example, \"classifyTriangle(int,int,int)\"", required = true)
    protected String methodSignature = "";

    @Argument(alias = "s", description = "Seed")
    protected Integer seed = 123;

    @Argument(alias = "n", description = "Number of steps")
    protected Integer numSteps = 100;
```

```

    @Argument(alias = "d", description = "Top directory")
    protected File packageDir;

    @Argument(alias = "c", description = "Class name")
    protected String className;

    @Argument(alias = "cp", description = "Classpath")
    protected String classPath;

    @Argument(alias = "t", description = "Test class name")
    protected String testClassName;

    @Argument(alias = "et", description = "Edit type: this can be a member of the EditType
    enum (LINE,STATEMENT,MATCHED_STATEMENT,MODIFY_STATEMENT); the fully qualified name of a
    class that extends gin.edit.Edit, or a comma-separated list of both")
    protected String editType = EditType.LINE.toString();

    /**
     * allowed edit types for sampling: parsed from editType
     */
    protected List<Class<? extends Edit>> editTypes;

    @Argument(alias = "ff", description = "Fail fast. "
            + "If set to true, the tests will stop at the first failure and the next patch
    will be executed. "
            + "You probably don't want to set this to true for Automatic Program Repair.")
    protected Boolean failFast = false;

    protected SourceFile sourceFile;
    protected Random rng;
    InternalTestRunner testRunner;

    // Constructor parses arguments
    LocalSearch(String[] args) {
        Args.parseOrExit(this, args);
        editTypes = Edit.parseEditClassesFromString(editType);

        this.sourceFile = SourceFile.makeSourceFileForEditTypes(editTypes, this.filename.
    toString(), Collections.singletonList(this.methodSignature));

        this.rng = new JDKRandomBridge(RandomSource.MT, Long.valueOf(seed));
        if (this.packageDir == null) {
            this.packageDir = (this.filename.getParentFile() != null) ? this.filename.
    getParentFile().getAbsoluteFile() : new File(System.getProperty("user.dir"));
        }
        if (this.classPath == null) {
            this.classPath = this.packageDir.getAbsolutePath();
        }
        if (this.className == null) {
            this.className = FilenameUtils.removeExtension(this.filename.getName());
        }
        if (this.testClassName == null) {
            this.testClassName = this.className + "Test";
        }
        this.testRunner = new InternalTestRunner(className, classPath, testClassName, failFast
    );
    }

    // Instantiate a class and call search
    public static void main(String[] args) {
        LocalSearch localSearch = new LocalSearch(args);
        localSearch.executeSearch();
    }

    // Apply empty patch and return execution time
    private long timeOriginalCode() {
```

```
        Patch emptyPatch = new Patch(this.sourceFile);
        UnitTestResultSet resultSet = testRunner.runTests(emptyPatch, WARMUP_REPS);

        if (!resultSet.allTestsSuccessful()) {
            if (!resultSet.getCleanCompile()) {
                Logger.error("Original code failed to compile");
            } else {
                Logger.error("Original code failed to pass unit tests");
                for (UnitTestResult testResult : resultSet.getResults()) {
                    Logger.error(testResult);
                }
            }
            System.exit(0);
        }

        return resultSet.totalExecutionTime() / WARMUP_REPS;
    }

    // Apply empty patch and return memory consumption
    private long memoryOriginalCode() {
        Patch emptyPatch = new Patch(this.sourceFile);
        UnitTestResultSet resultSet = testRunner.runTests(emptyPatch, WARMUP_REPS);

        if (!resultSet.allTestsSuccessful()) {
            if (!resultSet.getCleanCompile()) {
                Logger.error("Original code failed to compile");
            } else {
                Logger.error("Original code failed to pass unit tests");
                for (UnitTestResult testResult : resultSet.getResults()) {
                    Logger.error(testResult);
                }
            }
            System.exit(0);
        }

        return resultSet.totalMemoryUsage() / WARMUP_REPS;
    }

    // Method to get the memory consumption corresponding to a given execution time
    private long memoryForTime(long time) {
        Patch patchForTime = new Patch(this.sourceFile);
        UnitTestResultSet resultSet = testRunner.runTests(patchForTime, WARMUP_REPS);

        long startTime = System.nanoTime();
        while (System.nanoTime() - startTime < time) {
            // Running the patch for the specified time
        }

        return resultSet.totalMemoryUsage() / WARMUP_REPS;
    }

    // Method to get the execution time corresponding to a given memory consumption
    private long timeForMemory(long memory) {
        Patch patchForMemory = new Patch(this.sourceFile);
        UnitTestResultSet resultSet = testRunner.runTests(patchForMemory, WARMUP_REPS);

        long startTime = System.nanoTime();
        while (resultSet.totalMemoryUsage() / WARMUP_REPS < memory) {
            // Running the patch until it reaches the specified memory consumption
        }

        return System.nanoTime() - startTime;
    }

    // Simple local search
    private void executeSearch() {
        Logger.info(String.format("Localsearch on file: %s method: %s", filename,
```

```java
methodSignature));

        // Time original code
        long origTime = timeOriginalCode();
        Logger.info("Original execution time: " + origTime + " ns");

        // Memory consumption of original code
        long origMemory = memoryOriginalCode();
        Logger.info("Original memory consumption: " + origMemory + " Mbytes");

        // Start with empty patch
        Patch bestPatch = new Patch(this.sourceFile);
        long bestTime = origTime;
        long bestMemory = origMemory;

        // Initializing the best score to be maximum (worst case)
        double bestScore = Double.MAX_VALUE;

        for (int step = 1; step <= numSteps; step++) {
            Patch neighbour = neighbour(bestPatch);

            // Time execution for the neighbor
            UnitTestResultSet testResultSet = testRunner.runTests(neighbour, 1);

            String msg;

            if (!testResultSet.getValidPatch()) {
                msg = "Patch invalid";
            } else if (!testResultSet.getCleanCompile()) {
                msg = "Failed to compile";
            } else if (!testResultSet.allTestsSuccessful()) {
                msg = "Failed to pass all tests";
            } else {
                long newTime = testResultSet.totalExecutionTime();
                long newMemory = testResultSet.totalMemoryUsage();

                // Normalize the time and memory consumption (assuming smaller is better for both)
                double normTime = (double)newTime / origTime;
                double normMemory = (double)newMemory / origMemory;

                // Sum of normalized time and memory can be your score
                double newScore = normTime + normMemory;

                if (newScore < bestScore) {
                    bestPatch = neighbour;
                    bestScore = newScore;
                    bestTime = newTime;
                    bestMemory = newMemory;
                    msg = String.format("New best score: %.2f, with time: %d (ns) and memory: %d (Mbytes)", bestScore, bestTime, bestMemory);
                } else {
                    msg = String.format("Score: %.2f, with time: %d (ns) and memory: %d (Mbytes)", newScore, newTime, newMemory);
                }
            }

            Logger.info(String.format("Step: %d, Patch: %s, %s", step, neighbour, msg));
        }

        System.out.println("\n");

        Logger.info(String.format("Finished. Best time: %d (ns), Speedup (%%): %.2f, Best memory consumption: %d Mbytes, Memory reduction: %.2f%%, Patch: %s",
                bestTime,
                100.0 * ((origTime - bestTime) / (1.0 * origTime)),
                bestMemory,
```

```
                100.0 * ((origMemory - bestMemory) / (1.0 * origMemory)),
                bestPatch));

        bestPatch.writePatchedSourceToFile(sourceFile.getRelativePathToWorkingDir() + ".
    optimised");
    }



    /**
     * Generate a neighboring patch, either by deleting an edit or adding a new one.
     *
     * @param patch Generate a neighbor of this patch.
     * @return A neighboring patch.
     */
    Patch neighbour(Patch patch) {
        Patch neighbour = patch.clone();

        if (neighbour.size() > 0 && rng.nextFloat() > 0.5) {
            neighbour.remove(rng.nextInt(neighbour.size()));
        } else {
            neighbour.addRandomEditOfClasses(rng, editTypes);
        }

        return neighbour;
    }
}
```

A.5 The Triangle java code

```
import java.util.Arrays;

public class Triangle {
    static final int INVALID = 0;
    static final int SCALENE = 1;
    static final int EQUALATERAL = 2;
    static final int ISOCELES = 3;

    public static int classifyTriangle(int a, int b, int c) {

        // Consume more memory by creating a large array
        int[] largeArray = new int[1000000];
        Arrays.fill(largeArray, 0);

        delay();

        // Sort the sides so that a <= b <= c
        if (a > b) {
            int tmp = a;
            a = b;
            b = tmp;
        }

        if (a > c) {
            int tmp = a;
            a = c;
            c = tmp;
        }

        if (b > c) {
            int tmp = b;
            b = c;
            c = tmp;
        }
```

```

        if (a + b <= c) {
            return INVALID;
        } else if (a == b && b == c) {
            return EQUALATERAL;
        } else if (a == b || b == c) {
            return ISOCELES;
        } else {
            return SCALENE;
        }

    }

    private static void delay() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {

        }
    }
}
```

A.6 The `Greatest Common Divisor (GCD)` java code

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;

public class GCD {

    static final int INVALID = -1;

    public static int findGCD(int a, int b, int c) {

        // Consume more memory by creating a large ArrayList with random values
        ArrayList<Integer> largeList = new ArrayList<>(1000000);
        Random random = new Random();
        for (int i = 0; i < 1000000; i++) {
            largeList.add(random.nextInt());
        }

        complexComputation(); // Introduce a complex computation

        // Ensure that a, b, and c are positive
        if (a <= 0 || b <= 0 || c <= 0) {
            return INVALID;
        }

        // Increasing the loop size to consume more execution time
        for (int i = 0; i < 10000; i++) {
            double temp = Math.sqrt(i) * Math.log(i + 1); // More complex unused computation
        }

        // Find the GCD of a and b
        int gcdAB = gcd(a, b);

        // Find the GCD of gcdAB and c
        return gcd(gcdAB, c);
    }

    private static int gcd(int a, int b) {
        if (b == 0) {
            return a;
        }
        return gcd(b, a % b);
```

```
    }

    private static void complexComputation() {
        // Increasing sleep time and adding more computations
        try {
            Thread.sleep(200); // Increased sleep time
        } catch (InterruptedException e) {

        }

        // Perform complex computations
        double sum = 0;
        for (int i = 0; i < 1000; i++) {
            for (int j = 0; j < 1000; j++) {
                sum += Math.sin(i) * Math.cos(j);
            }
        }
    }
}
```

A.7 The Rectangle java code

```
import java.util.Arrays;

public class Rectangle {

    static final int INVALID = 0;
    static final int RECTANGLE = 1;
    static final int SQUARE = 2;

    public static int classifyRectangle(int a, int b, int c, int d) {
        // Consume more memory by creating a large array
        int[] largeArray = new int[1000000];
        Arrays.fill(largeArray, 0);

        // Adding a delay
        try {
            // Pausing for 2 seconds (2000 milliseconds)
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // A rectangle or square will be invalid if any side length is less than or equal to 0
        if(a <= 0 || b <= 0 || c <= 0 || d <= 0){
            return INVALID;
        }

        // If all sides are equal
        else if(a == b && b == c && c == d){
            return SQUARE;
        }

        // If opposite sides are equal
        else if(a == c && b == d){
            return RECTANGLE;
        }

        // If the given sides can't form a rectangle or square
        else{
            return INVALID;
        }
    }
}
```

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. 11

[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. 7

[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. iv, 13, 14, 15, 16

[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. 13

[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. 13

[Calero and Piattini, 2017] Calero, C. and Piattini, M. (2017). Puzzling out software sustainability. *Sustain. Comput. Informatics Syst.*, 16:117–124. 3

[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. 14

[Couto et al., 2017] Couto, M., Pereira, R., Ribeiro, F., Rua, R., and Saraiva, J. (2017). Towards a green ranking for programming languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, pages 1–8. 18

[dos Reis et al., 2020] dos Reis, J. P., e Abreu, F. B., de Figueiredo Carneiro, G., and Anslow, C. (2020). Code smells detection and visualization: A systematic literature review. *CoRR*, abs/2012.08842. 38

[Fontana et al., 2016] Fontana, F. A., Mäntylä, M. V., Zanoni, M., and Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empir. Softw. Eng.*, 21(3):1143–1191. 38

[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. 7

[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. 11

[Lima et al., 2016] Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., and Fernandes, J. P. (2016). Haskell in green land: Analyzing the energy behavior of a purely functional language. In *2016 IEEE 23rd international conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 517–528. IEEE. 18

[Liu and Zhang, 2017] Liu, X. and Zhang, C. (2017). The detection of code smell on software development: a mapping study. In *2017 5th International Conference on Machinery, Materials and Computing Technology (ICMMCT 2017)*, pages 560–575. Atlantis Press. 37

[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. 9, 10, 11

[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. 13

[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. 7

[Noureddine and Rajan, 2015] Noureddine, A. and Rajan, A. (2015). Optimising energy consumption of design patterns. In Bertolino, A., Canfora, G., and Elbaum, S. G., editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 623–626. IEEE Computer Society. 5, 6

[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. 9, 10

[Paradis et al., 2021] Paradis, C. V., Kazman, R., and Tamburri, D. A. (2021). Architectural tactics for energy efficiency: Review of the literature and research roadmap. In *54th Hawaii International Conference on System Sciences, HICSS 2021, Kauai, Hawaii, USA, January 5, 2021*, pages 1–10. ScholarSpace. 5, 6

[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. 11

[Pereira et al., 2017] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017). Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, pages 256–267. 18

[Pereira et al., 2021] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. (2021). Ranking programming languages by energy efficiency. *Sci. Comput. Program.*, 205:102609. 3

[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. 12

[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. 9, 10, 20

[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. 5, 7

[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. 7

[Tsantalis et al., 2018] Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2018). Ten years of jdeodorant: Lessons learned from the hunt for smells. In Oliveto, R., Penta, M. D., and Shepherd, D. C., editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 4–14. IEEE Computer Society. 37

[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. 12, 13