UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Energy Aware Programming

Afonso Vaz dos Santos Silva Carreira

Mestrado em Engenharia Informática

Versão Provisória

Dissertação orientada por:
Prof. Doutor Alcides Miguel Cachulo Aguiar Fonseca
Prof. Doutor Wellington Oliveira Júnior

2025

Acknowledgments

I want to express my gratitude to my family, parents, sister and grandparents for their constant encouragement, assistance and love towards me throughout my academic career. Their confidence in me has at all times been a source of inspiration.

I would also like to thank my supervisors, Professor Alcides and Professor Wellington, for their perceptive feedback, guidance, and patience throughout the writing of this thesis. Their mentoring was integral to this piece of work's success.

Dedicatória.

Resumo

Nos últimos anos, o uso e a gestão da energia tornaram-se uma questão global. Está em curso a procura de energias renováveis que reduzam o impacto ecológico no nosso planeta. No entanto, estas alternativas não são tão acessíveis nem tão consistentes como as opções tradicionais. Existem muitas áreas em que é possível reduzir a pegada energética, e o sector da informação e tecnologia é um deles. Poupar energia ao nível de aplicações é importante para todas as categorias de dispositivos eletrónicos desde dispositivos móveis a centros de dados. No entanto, estima-se que, em 2020, cerca de 7% da energia elétrica global tenha sido utilizada para fins de tecnologia da informação e comunicação. Com a crescente procura por novas tecnologias, prevê-se que esse número continue a aumentar nos próximos anos. Esta tendência mostra que tem que existir um cuidado maior por parte dos programadores na fase de desenvolvimento de código que seja energeticamente eficiente. No entanto, quando de facto tentam procurar formas de reduzir os custos energéticos dos seus programas, e recorrem a *websites* ou grandes modelos linguísticos, e muitas vezes as respostas obtidas podem não estar completamente corretas. Existe ainda uma necessidade de ferramentas de fácil utilização que permita que os programadores rapidamente e sem grande conhecimento na área da energia consigam perceber a energia utilizada pelos seus programas. Esta tese tem como objectivo aumentar a consciencialização dos programadores para a energia dos seus programas, apresentando uma *framework* que facilita a obtenção de modelos que realizam a previsão de energia. Ela segue um fluxo estruturado que começa com a geração de vários benchmarks usando várias categorias de coleções ou classes definidas pelo programador. Esses benchmarks são então submetidos a um análise que permite obter perfis energéticos para treinar modelos para cada método na classe alvo. Os modelos resultantes são integrados numa extensão de ambiente de desenvolvimento integrado capaz de identificar o consumo de energia dos métodos nos programas e apresentar essas informações rapidamente aos programadores, permitindo-lhes tomar decisões informadas no desenvolvimento de aplicações. Essa maior consciencialização permitirá que os programadores compreendam o impacto geral das suas escolhas de programação no consumo e na eficiência energética. Para ser possível obter os modelos que consigam prever a energia de programas, é primeiro necessário ter uma quantidade considerável de dados. Para isso, foi desenvolvido um gerador de benchmarks, com o objetivo de produzir automaticamente um grande número de exemplos com variações relevantes. Este gerador permite criar benchmarks para várias coleções de linguagem Java ou para programas personalizados. A geração é feita com base na coleção e no método especificado, produzindo múltiplas combinações através da variação dos tipos de variáveis

e dos valores de entrada. Cada benchmark gerado foi estruturado para analisar exclusivamente um método específico, estando otimizado para a sua avaliação através de testes de desempenho. A segunda etapa consiste em utilizar todos os benchmarks gerados para analisar individualmente cada um, com o objetivo de obter os perfis de consumo de energia associados a cada método em estudo. Esta etapa consistiu em construir um processo eficiente de recolha dos dados de consumo energético dos benchmarks, minimizando leituras desnecessárias e ruído, tendo em conta as limitações inerentes à medição de energia em aplicações. Para a recolha dos dados, foi utilizada a ferramenta PowerJoular, que permite monitorizar o consumo energético de forma precisa durante a execução dos programas. Para ilustrar o funcionamento da framework neste trabalho foram recolhidos dados para as coleções `List` e `Map` da linguagem Java, que serviram de base para o treino e validação dos modelos. Após a obtenção dos perfis de energia, foi possível treinar modelos de aprendizagem automática supervisionada com os dados recolhidos. Nesta fase, foram comparados diversos algoritmos, tendo-se optado pelo PySR, uma ferramenta que gera um conjunto de expressões matemáticas ordenadas da mais simples para a mais complexa. Esta abordagem permite equilibrar entre desempenho e interpretabilidade, oferecendo modelos transparentes, ao contrário dos modelos de caixa preta, e facilitando a visualização da expressão utilizada para prever o consumo energético. Por fim foi construida uma extensão para o VSCode que integra os modelos previamente treinados. Esta ferramenta combina análise estática com aprendizagem automática para realizar previsões de energia de programas. Esta combinação permite que o cálculo da energia utilizada seja bastante rápido, evitando a necessidade de executar programas, e aumentando a consciencialização dos programadores para a eficiência energética dos seus programas. Os resultados obtidos mostram que a framework consegue gerar modelos para programas personalizados necessitando apenas de alterações mínimas. A ferramenta de previsão de energia revelou-se útil para identificar variações relativas no consumo energético, à medida que os valores de entrada aumentam, observa-se um aumento correspondente nos valores de energia. No entanto, ao nível de valores absolutos, a precisão das previsões depende diretamente do desempenho dos modelos treinados, sendo necessária uma melhoria nestes para obter resultados mais precisos.

Em resumo, esta tese contribui para o campo da programação com consciência energética, combinando aprendizagem automática e análise estática para permitir uma previsão energética precisa e acessível. No entanto, a investigação e o desenvolvimento contínuos são essenciais para melhorar ainda mais a precisão do modelo, expandir a cobertura de funcionalidades e apoiar uma variedade mais ampla de cenários de programação.

Abstract

In recent years, energy use and management have become global concerns, with a growing search for renewable sources to reduce the ecological footprint. However, these alternatives are not always as accessible or reliable as traditional energy sources. The information and communication technology (ICT) sector is one area where significant reductions in energy consumption are possible. In 2020, around 7% of global electricity consumption was attributed to ICT, and this number is expected to rise due to increasing technological demand. Consequently, there is a growing need for developers to adopt energy-aware programming practices.

This thesis aims to raise developers' awareness of the energy consumption of their applications by presenting a framework for generating energy prediction models. The framework follows a structured process, beginning with the automatic generation of Java benchmark programs for specific collections methods or custom methods. These benchmarks are then analyzed to obtain energy profiles, which are used to train machine learning models capable of estimating energy consumption. The final models are then integrated in a VSCode extension tool that combines static analysis and machine learning to predict the code energy consumption, this allows to obtain an estimation of how much energy the code uses, withiout the need to execute it.

The results show that the tool can support energy prediction for customized programs and highlight the relationship between input size and energy usage. While the relative accuracy is promising, further work is needed to improve the precision of absolute predictions and expand the applicability of the framework.

Keywords: Green computing, Static code analysis, Energy prediction, Sustainable software, Energy-aware programming

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, the use and management of energy has become a global issue. The search is on for renewable energies that reduce the ecological impact on our planet. However, these alternatives are neither as cheap nor as consistent as traditional options. There are many areas in which that may reduce their energy footprint, and the IT sector is one of them [1, 2].

Saving energy in programs is crucial for the operation of certain devices, such as mobile phones or IoT devices, so certain techniques need to be applied in order to reduce the energy of a program. For mobile devices, companies such as Google and Apple have developed tools [3, 4, 5, 6] to help save energy and while running the applications techniques are already used to save the battery when necessary, but for systems that do not use batteries, such as servers, energy is rarely taken into account when developing a program. This lack of concern or awareness on the part of developers, although it appears to have a small impact, turns out to be quite significant. In 2020 around 7% of global electricity use was due to information and communications technology, with an anticipated rise in line with the growing demand for new technologies [7]. This trend has become even more significant with the increased use of artificial intelligence [8], large scale models such as ChatGPT, Gemine, DeepSeek, which require significant computing resources to train and run. These energy-intensive processes contribute significantly to global energy consumption and carbon emissions, raising environmental concerns as AI adoption continues to grow. For instance, training the GPT-3 model required 1,287 MWh of energy, equivalent to the annual energy usage of approximately 117 U.S. households, and produced 552 metric tons of CO2, comparable to the emissions from driving 120 cars for a year. With the release of GPT-4 and ongoing development of even more advanced models, these numbers are expected to rise, further amplifying their environmental footprint. The significant energy demands of data centers lead to considerable heat generation, requiring Heating, Ventilation, and Air Conditioning (HVAC) systems to ensure stable operations. Remarkably, HVAC systems consume approximately 33% of a data center's total energy, with another 18% dedicated to Computer Room Air Conditioning units. Servers, which are integral to data center functionality, account for 45% of the energy usage, even without factoring in AI-driven tasks and complex modeling workloads [9].

Some key reasons to prioritize energy efficiency in software, whether for mobile systems or data center applications, include:

- The dependence of mobile devices on batteries. All mobile devices rely on their batteries, so the software they run needs to make the best use of resources to conserve battery power.

- Reducing operating costs in data centers. It is crucial to reduce the operating cost of data centers by using energy-efficient programs. This reduction results in economic benefits for companies and contributes positively to environmental sustainability.

- Reducing energy consumption has a positive impact on our environment by saving energy that can be used more efficiently elsewhere.

When developing a program, most of the time developers optimize for the time the program takes to complete, or the memory it uses, and not so often take into account the energy it uses [2]. In the cases where developers actually want to improve the energy efficiency of the code, they normally have difficulties and seek help, relying on blogs, websites and YouTube videos, which in most cases lack empirical evidence, leading to perceptions of improvement rather than measured benefits [10].

Recently, developers have also started consulting Large Language Models, such as GitHub Copilot or ChatGPT, to get suggestions for writing energy-efficient code. These models can offer advice on optimized algorithms, data structures, and design patterns, which could lead to more sustainable software development [11, 12]. However, their guidance is not yet fully reliable when it comes to energy efficiency [13], as the models still, on average, output code that is less energy efficient than human written solutions.

This is due to a lack of knowledge and guidelines, because understanding the energy usage of a program and how to make it more efficient is not trivial, as running the same program multiple times will output different values each time and even if the execution time of the program is reduced, is not guaranteed to also reduce energy consumption. Because of its difficulty, there is still a need for tools that can help with this task [14].

Also, most current tools can measure the energy of programs and applications as they run (e.g., Android Studio Power Profiler [6]), but this usually requires extra steps that many developers may not have the time or inclination to take, so there is a need for a tool that can help the developer without the need for extra effort [10].

1.2 Objectives

This thesis proposes a framework with the objective of increasing energy awareness of developers. The framework follows a structured pipeline that begins by generating multiple benchmarks using various collection types or developer-defined classes. These benchmarks are then energy-profiled to train predictive models for each method within the target class. The resulting models are integrated into an IDE extension capable of identifying the energy consumption of methods

in programs and presenting this information quickly to programmers, enabling them to make informed decisions in software design. The goal is to create a framework with an approach that can be reproducible for obtaining models capable of performing energy prediction statically. The framework is divided into modular components that can be improved independently to optimize model performance.

To create this framework, it was essential to understand the current state of the art, including the techniques previously used and the tools currently in use. The tool employs static analysis techniques to identify which instructions are utilized, and through models from previously collected data, indicate the estimated energy levels of the program's execution. The models will be trained using energy data collected from low-level library functions. More complex functions are built on the basis of function composition, which means that, based on the estimated consumption of low-level functions, we can generalize our estimates to more complex functions and ultimately to the program as a whole.

This increased awareness will enable them to understand the overall impact of their coding choices on energy consumption and efficiency.

1.3   Contributions

This thesis presents a framework that automates the generation, execution, energy profiling, and modeling of software programs to enable energy-aware static analysis. The main contributions of this work are:

- **Benchmark Generation:** A module that automatically generates a large set of Java benchmarks targeting specific methods. It systematically varies method inputs and parameter types to enable broad coverage and controlled diversity.

- **Energy Profiling:** An orchestration module that compiles and executes each generated benchmark while measuring energy consumption at runtime. It captures power usage signals at the method level to build accurate energy profiles.

- **Model Training:** A machine learning pipeline that trains predictive models on the collected energy profiles. These models learn to estimate the energy behavior of program methods based on static and dynamic features.

- **Energy Prediction with Static Analysis and Machine Learning:** The tool combines static program analysis with machine learning techniques to predict the energy usage of Java applications without requiring program execution. This approach allows for fast estimations of energy consumption, and helps with the challenge of the non-deterministic nature of energy behavior in software.

- **User-Friendly Interface:** A simple graphical user interface (GUI) was developed to enable user interaction with the models. It includes sliders to change input features and observe how these changes affect the predicted energy usage.

- **Method-Based Energy Calculation:** The tool predicts energy at the method level. If one method calls others, its energy is estimated by summing the predictions of those calls. For example, if `foo()` calls `bar()` and `baz()`, its energy is based on the models trained for those two operations. This approach makes the predictions more modular and easier to reuse across different programs.

- **Support for Future Improvements:** The framework is built to be flexible and adaptable, allowing for easy integration of new features, or learning algorithms. This makes it a good foundation for future work in the energy-aware software area.

1.4   Document Structure

This document is organized as follows:

- Chapter 2 - introduces key concepts necessary to fully understand the report. It discusses the challenges of predicting and measuring energy consumption in programs, explores various energy tools and machine learning techniques, and provides an overview of static and dynamic analysis, highlighting their relevance to this work.

- Chapter 3 - contains the initials solutions proposed to the theme of energy aware programming, how they changed during the years, and what the most recent tools do. And comparing with the proposed tool in this work.

- Chapter 4 - explains in detail the existing problem and what is the solution, and a detailed explanation on how the solution is built.

- Chapter 5 - reports on the experiments made, and the obtained results.

- Chapter 6 - summarizes the work completed to date.

- Chapter 7 - outlines future research directions.

Chapter 2

Background

This section provides an overview of the essential concepts and foundational knowledge necessary for understanding the methods and approaches used throughout this thesis. A solid understanding of these concepts is necessary to fully comprehend the work presented. The section begins with a discussion on benchmarking, inlcuding the best practices for performing benchmarks. It then explores energy profiling, which involves benchmarking but focused on energy measurements. It explains the energy tools capable of performing the energy measurements, how they work, what are the advantages and disadvantages. Additionally, it introduces the concepts of static and dynamic analysis explaining the differences and each is applied. The Language Server Protocol concept is also examined as it has relevance in the context of this thesis. Finally, it has a brief explanation on machine learning and how it can be used in our project.

2.1 Benchmarking

Benchmarking is the process of measuring the performance of a program or system by running a series of tests under controlled conditions. This typically involves evaluating factors such as execution time, memory usage, and energy consumption. The purpose of benchmarking is to gain a clear understanding of how the software behaves in different scenarios, to compare different implementations, and to identify potential areas for improvement.

In this project, the primary goal of benchmarking is to accurately measure the energy consumption of Java applications, while taking into account the factors that can affect these measurements.

When performing benchmarking (i.e. measuring time, energy, memory, etc.) it is important to have in mind some important information that can affect the measurements. First it matters to know how the programming language being used works, and how it can affect the measurements. It is important to understand which noises can be avoided and which cannot.

As an example, benchmarks in Java, which is the primary programming language used in this project, will have its constraints. Performing benchmarks in Java has its adversities, and needs careful approach when needed. Several papers have studied how to perform better benchmarks [15, 16] details important aspects on how to correctly benchmark Java applications, and how to avoid common pitfalls. It is essential to understand the non-deterministic problems that may arrive when

benchmarking in Java:

- **Just-In-Time (JIT) compilation**: Automatically performs optimizations in during the compilation, which can affect measurements. Example: In the loop presented in the Listing 1, the optimization done during the compilation, might understand that the loop is unnecessary and just replace the variable sum with `int sum = 100;`, avoiding the loop. Basically, the JIT compiler may detect that a loop performs predictable work and optimize it away entirely, especially if the results are unused or can be computed ahead of time.

```java
int sum = 0;
for (int i = 0; i < 100; i++) {
    sum++;
}
System.out.println(sum);
```

Listing 1: Example for loop that can be optimized by the JIT compiler.

- **Garbage Collection (GC)**: Garbage collection can occur unpredictably, which may impact the accuracy of measurements. For time and energy measurements, it often results in increased values, while for memory measurements, the effect depends on whether garbage collection occurs before or after the measurement period.

- **Thread scheduling**: It is also unpredictable when threads stop might have different schedules, resulting in different interactions.

- **Java Virtual Machine (JVM)**: Different JVM implementations or configurations (e.g., heap size, GC algorithms, JIT strategies) can result in considerable performance differences. Even the same JVM may behave differently across runs due to internal optimizations and adaptive behaviors. Additionally, JVM startup time can be a problem in performance measurements, especially in short processes, as it introduces overhead not representative of steady-state behavior.

Some of these properties cannot be avoided, however there are some procedures that can be taken into consideration to avoid error in the measurements. To avoid JIT optimizations, harder and complex examples must be used in order to avoid optimizations, for example instead of filling a list in a for loop using the index values, the values added can be random, which will avoid optimizations, as the compiler cannot predict the next number in the iteration. This proves to be effective in energy measurement, because if it is needed to measure the energy consumption of the add method of a list collection, now the process can actually be measured avoiding optimizations, if it was optimized it would be too fast and would not be valid for energy measurement. The

JVM start up can be avoided by starting the program without measuring anything and just start the measurement when the computation method is ready to be started and not for the whole machine. Avoiding reading JVM startup, warm up, even unnecessary computations, and focusing only on the necessary parts. Garbage collector can be harder, while `System.gc()` can be used to suggest a garbage collection cycle, the JVM is free to ignore this request. As such, it is not a reliable mechanism to control GC timing. An alternative is to increase the memory the JVM can use, so the GC is called fewer times.

It is also important to note that:

- **Use multiple JVM invocations**: Run benchmarks in new JVM instances to capture variations and prevent biased warm-up effects.

- **Use multiple JVMs and versions**: Different JVM implementations and versions may apply different optimizations, affecting performance outcomes.

- **Avoid cherry-picking values**: Instead of reporting only the best or worst results, use the median or mean along with variability.

- **Vary hardware configurations**: Using different hardware will have different results.

- **Test with multiple heap sizes**: Garbage collection behavior and performance can change depending on the heap size. Test different benchmarks with a range of heap sizes, starting from the minimum required.

- **Use realistic, diverse workloads**: Use real-world applications instead of microbenchmarks to better reflect practical behavior.

- **Avoid background system noise**: When running the benchmarks, preferably only have the necessary processes running, to reduce measurement noise.

When building the tool, it is necessary to gather data from the Java programs, which requires benchmarking best practices. Although it is not possible to apply all of them due to time and complexity constraints, it is always helpful to keep in mind what it takes to create a good benchmark. In this project, the techniques applied included using multiple JVM invocations, avoiding cherry-picking by reporting average values, minimizing background system noise, excluding JVM startup time, and focusing measurements on the relevant computation phases.

2.2 Energy Profiling

To fully understand the processes involved in this work, it is important to first understand what energy profiling is, as it will be frequently referenced and used later.

Energy profiling is the systematic process of measuring, monitoring, and analyzing the power consumption of a system, using specialized software or hardware tools to collect detailed power

consumption data. This data can be collected from different parts of a system, including applications, processes, and specific snippets of code, to provide insight into how different components and activities contribute to overall energy consumption. Through this type of power consumption profiling, developers are able to identify inefficiencies and optimize software to improve energy efficiency. This makes it very important for extending battery life in mobile devices, reducing operating costs in data centers, and also minimizing environmental impact through energy consumption. Energy profiling is a step toward making informed decisions to develop more sustainable and cost-effective computing solutions.

Energy profiling bridges the gap between traditional performance metrics and energy consumption, offering developers critical insights into how their software impacts system efficiency. When programming, developers usually consider the time it takes to complete a program, or the response time from client to server, or the amount of memory it uses. Most don't have an idea of how much energy their program consumes, or how much it can consume in certain cases, and getting this idea is not as trivial as it seems. To get this awareness, measurements could be made, but they are also difficult to get compared to measuring the time a program takes, which is checking the differences in timestamps, or understanding the memory usage. As for measuring energy, the same program can produce different values due to its non-deterministic nature, meaning that results are often expressed as a range of possible values. Also, reducing the execution time of a program does not guarantee that power consumption will follow [17]. To obtain the measurements it is necessary to use software based tools that can facilitate this process, sometimes at the cost of less accurate readings or hardware devices for accurate values.

To perform hardware-based measurements, a power monitor or power meter device [18, 19] must be used to obtain the precise values that a system uses when connected to the electrical grid. These devices measure the power drawn from the grid to the machine, but they usually measure the power consumed by the entire system rather than by specific pieces of hardware, which means they have low granularity. Achieving granular measurements, such as isolating power consumption for specific components like the CPU or RAM, requires a more complex setup to avoid reading unnecessary power consumption. This approach is not suitable for developers who only want to analyze the energy performance of their programs, which shows the difficulty of measuring energy consumption.

An alternative with minimal overhead is to use software-based tools. These tools are typically easier to use, but may not provide values as accurate as hardware-based measurements. However, they are more versatile and can be implemented or modified to meet the user's needs.

2.3   Energy Tools

Understanding and optimizing energy consumption in software requires specialized tools capable of measuring and analyzing power usage. These tools provide valuable insights into how programs consume energy during execution, enabling developers to identify inefficiencies and optimize performance.

At the start some tools were tested in order to see the one that best suited the needs of the project:

Intel RAPL (Running Average Power Limit) [20] is a tool for monitoring power consumption. It utilizes Model-Specific Registers (MSRs), which are used for program execution tracing, performance monitoring, and toggling CPU features. These registers can store the total energy usage of the CPU and memory, allowing it to be read and analyzed. Most software based tools rely on RAPL to measure energy consumption, since it is pretty accurate and is widely available in most CPUs.

Perf [21] is a Linux tool primarily designed for analyzing application performance characteristics rather than precise energy measurement. While it can provide some energy-related metrics, its measurements tend to be imprecise. In this context, Perf was used mainly to get a rough idea of energy consumption and to serve as an alternative when more accurate tools were unavailable.

Powertop [22] was also tested, but it could only perform energy measurements on laptops, as it relies on battery drain data to calculate energy consumption. Since this approach does not align with our specific requirements, we considered Powertop as a last-resort option.

JoularJx [23] is an energy measurement tool capable of measuring the consumption of Java programs and its methods. However, it is not as precise as other tools as it requires to measure the entire start of the JVM and whole functions instead of small code blocks.

PowerJoular [23] is an open source tool, capable of measuring energy, from the CPU and GPU, using the Intel RAPL power data through the Linux powercap interface it can read the energy from the CPU and for the GPU it uses NVIDIA SMI to directly read the power consumption. To read the power consumption of specific processes, PowerJoular monitors the CPU cycles and utilization of each process. By knowing the total power consumption of the CPU through the RAPL interface, it can calculate the power usage of individual processes based on their CPU utilization. It is build in ADA, that is considered one of most energy efficient programming languages [24], and it can monitor applications by name or PID. In this work, it will be necessary to measure the energy consumption of programs, methods, and specific code snippets. To achieve this, PowerJoular will be employed as the primary tool for energy profiling.

Experiment-Runner [25] is a framework built in python made to facilitate experiments, it is easily customizable and can be used with energy measurement tools, such as PowerJoular, to monitor the power consumption of another process. While it offers robust support for energy-related experiments, some issues were identified during its use, which will be discussed in detail in Section 5.

Table 2.1: Comparison of energy measurement tools

Tool	Pros	Cons
Intel RAPL [20]	- Hardware-level accuracy - Widely supported in modern CPUs - Low overhead and high precision	- Not easily accessible - Cannot measure per-process energy directly
Perf [21]	- Standard Linux tool - Useful for performance + rough energy metrics	- Not designed for energy profiling - Low precision in energy consumption
Powertop [22]	- Easy to use - Good for overall system power stats on laptops	- Relies on battery drain - Ineffective on desktops or servers
JoularJx [23]	- Measures Java program energy - Method-level granularity possible	- Less precise - Only works with Java - Can only measure large code blocks
PowerJoular [23]	- Support for CPU and GPU - Supports per-process measurement - Built in energy-efficient ADA - Monitors by name or PID	- Multi-thread measurement is limited - Slightly more complex setup
Experiment Runner [25]	- Highly customizable - Integrates well with PowerJoular - Ideal for repeatable experiments	- Limited control over measurement internals, since it relies on a prebuilt framework instead of a custom solution - Some bugs and issues in practical use

2.4 Code static and dynamic analysis

Static analysis, as the name implies, is the process of analyzing the code statically, meaning it examines the code without executing it. By examining the code, tools that leverage static analysis tools can understand how the program will behave at runtime [26], this analysis often aims for soundness, meaning that if the tool catches an error, it means that the error really exists, there are no false negatives. However, this can come at the cost of producing false positives, where issues that are not actually problems are flagged, so it's important to keeps a balance between them. This analysis allows to check the entire source code and every path, much like compilers check syntax and types. Still, they can only predict some behaviors, as some can only be found when the program is executed, for example, by using dynamic analysis.

Dynamic Analysis is the process of analyzing a program while it is running, allowing for real-time observation of its behavior. This type of analysis leaves no doubt about memory usage, output, the path taken, how much time it took, among many other factors that are only clear when executing the program [26]. A good example of dynamic analysis is unit testing, which tries to cover as many code paths as possible with different inputs, to understand as much as possible

how the program works, and to find something that might be difficult to find with static analysis. However, because its nature, dynamic analysis can be time-consuming, it requires executing the program, which can be slow, especially for large applications or complex codebases. It also requires a lot of resources, such as memory and processing power, and it is not always possible to execute the program in all scenarios. For example, some programs may require specific hardware or software configurations that are not available in all environments. Additionally, dynamic analysis can only analyze the code that is executed during the run, which means that some parts of the code may not be covered by the analysis. For fast power estimation, static analysis is often preferable due to its speed and scalability. While it may be less accurate than dynamic analysis, it remains a viable option, particularly for large codebases with many dependencies. Static analysis avoids the need for code execution and is generally less resource-intensive, as discussed by Ernst in the context of program analysis trade-offs [26]. In addition, static analysis is more portable because its setup is much simpler than the more complex setup required for dynamic analysis. Developers may not have the time or the infrastructure to run the program just to get an average measure of energy consumption for a code snippet or program. Therefore, using static analysis to infer energy consumption makes sense in this context.

The use of static analysis implies the use of a parsing technique. This technique involves analyzing the syntactic structure of the provided code, respecting the rules of the language in which it is written. First, it is necessary to perform a lexical analysis to obtain the keywords, identifiers and tokens that the language contains. Then the parser uses these tokens together with the grammar rules of the programming language and outputs a tree. The output is an Abstract Syntax Tree (AST), which contains the logical structure of the code and allows further analysis. In the context of this work, this technique allows analyzing, for example, Java code and obtain its structure to find out which statements have been used.

The tool proposed in this work will primarily rely on static analysis to achieve its objectives. Static analysis, by examining the code without executing it, is particularly effective in providing early insights into potential energy inefficiencies during the development process. By evaluating all possible code paths and scenarios, it avoids the dependence on specific runtime conditions inherent in dynamic analysis, making it a powerful tool for identifying and addressing energy-related issues without the complexity of real-time monitoring.

However, to create the energy consumption dataset required to train machine learning models, this work will utilize dynamic analysis. This approach enables the collection of accurate and context-aware energy consumption measurements, which are crucial for building a reliable dataset to inform and enhance the energy optimization models. By combining the strengths of both static and dynamic analysis, this work aims to develop a comprehensive framework for energy-efficient software design.

2.5   Language Server Protocol

The Language Server Protocol (LSP) is an open protocol developed by Microsoft for separating language logic (such as code completion, diagnostic information, and symbol resolution) from the editor or development environment on which they are run. The objective of LSP is to allow programming language tools to be reused [27] in different code editors to gain better portability while also reducing the work required to support different environments.

The LSP runs on top of a client-server architecture. The language server, which is usually implemented in the language being analyzed, provides a number of features, such as syntax highlighting, error detection, semantic analysis, and navigation in the source. The client, usually embedded in the editor, talks over the JSON-RPC-based protocol to the server [28].

With LSP, language-specific behavior can be integrated in multiple editors without reimplementation of the underlying logic for each editor. For example, while in-plugin configuring of an environment such as IntelliJ IDEA or Eclipse generally requires deep integration of the in-plugin code with the specific platform's native APIs, an LSP implementation eliminates such complexity. If the editor is LSP-compatible, something which is the case for most modern editors, the editor is then able to operate with the language server without further configuration.

Within the context of the work, LSP is employed as the main technology for the development of a cross-platform language analysis backend. The language server is developed in Java, encapsulating the static analysis and model interaction logic, while the frontend is developed in web technologies (TypeScript and HTML) to provide the user interface within Visual Studio Code.

While LSP provides cross-editor support for language-related features, it does not account for UI integration into editors. For example, VSCode supports custom interfaces like WebView APIs, which allow extensions to show HTML and TypeScript information in the editor. However, other IDEs like IntelliJ IDEA or Eclipse use different UI integration schemes and don't have web-based interfaces supported in the same way. Therefore, while the language analysis component of the extension can be reused in editors supporting LSP, the UI may need to be redesigned or adapted for each environment.

2.6   Machine Learning

Using a machine learning (ML) model to estimate energy consumption can offer advantages over a traditional approach. While traditional approaches such as empirical estimates based on historical data may work, they may not be the best solution in this case due to the unpredictable behavior of energy. Using an ML algorithm can help identify more complex patterns and provide a highly accurate estimate while adapting to the arrival of new information.

To predict energy, an ML model will be used. It's important to understand how different ML algorithms work and which ones are best suited for the proposed project. There are several ML algorithms, and they fall into four main categories [29]: supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning.

The supervised ML requires labeled data for the model to train on. During training, the model has access to the input and output parameters, and it will try to match inputs to the correct outputs. It has two categories: classification, where it predicts discrete labels, such as whether a picture is a cat or a dog, and regression, where it predicts continuous values, for example, predicting the price of a house based on location, size, and so on, or, as in our project, predicting energy consumption. These models can be very accurate, but they can also make incorrect predictions for patterns they were not trained on. These techniques also are accompanied by some metrics to evaluate how well the models perform.

For *classification models*, some of the most important metrics include:

- **Accuracy:** Measures the percentage of correctly classified instances.

- **Precision:** Focuses on the correctness of positive predictions.

- **Recall (Sensitivity):** Evaluates how well the model detects actual positives.

- **F1 Score**: Balances precision and recall, useful when data is imbalanced.

- **ROC-AUC (Receiver Operating Characteristic - Area Under Curve):** Assesses classification performance across different thresholds.

On the other hand, some of the most important metrics for *regression models* are:

- **R² (R-squared or Coefficient of Determination):** Measures how well the model's predictions fit the actual data. A value close to 1 indicates a better fit.

- **MSE (Mean Squared Error):** Computes the average squared difference between predicted and actual values, penalizing large errors more heavily.

- **RMSE (Root Mean Squared Error):** The square root of MSE, offering a more interpretable error magnitude.

- **MAE (Mean Absolute Error):** Calculates the average absolute difference between predictions and actual values, making it less sensitive to extreme outliers.

In unsupervised ML, the model attempts to find patterns and relationships in the unlabeled data set. With this technique, it is possible to find similarities or clusters in the data, for example, to detect an anomaly in the data. This technique does not require the effort of acquiring labeled data, but also it will be harder to understand if the output is correct or not.

Semi-supervised learning, as the name implies, uses a combination of supervised and unsupervised learning techniques. This hybrid approach is very useful in real-world scenarios where parts of the data can be labeled and others can't, allowing for better predictions in the output.

Reinforcement learning, where the model receives different feedback for different tasks and uses this feedback to perform the tasks in the most optimal way. The feedback can be in the form of rewards or penalties so that the model can better understand whether it is doing the task correctly.

For example, a model playing a video game is rewarded for completing levels faster and penalized for failing the level. This method of learning allows for complex solutions to sequence-based problems, such as robotics or gaming. However, it can be time-consuming and computationally expensive.

There are some algorithms that can meet the proposed model requirements, such as: Linear regression, Tree-Based Models (e.g., Random Forest, Gradient Boosting), Neural Networks, Gaussian Processes, Support Vector Machines (SVM) and or leveraging Genetic Programming.

The most common approach in ML is to use a linear regression algorithm, which is the simpler to implement and very effective. It can predict a continuous output based on the input independent variables. Linear regression is computationally efficient, easy to implement, and works well when the relationship between the input features and the output is linear. However, its simplicity is also its limitation, as it struggles with nonlinear relationships and can underperform when the input features interact in complex ways. It is also sensitive to outliers, which can significantly skew the results.

Tree-based models rely on decision tree models, which are used for structuring decisions, where in each branch a decision is made based on some criteria, and the end of the branch contains the final output. Random forest is a tree-based model that combines multiple decision trees to build an accurate model. When a prediction is needed, all the decision trees provide a vote, in classification or an average in regression and the random forest combines them to give the final prediction. Each tree is trained in different subsets of the data. This algorithm provides high accuracy, robustness to overfitting and estimates the features' importance, however, it has a higher computational cost, more memory usage, and it can take more time to reach a prediction than other approaches.

Gradient boosting is also a tree-based model, it builds trees (weak learners) sequentially with each of them trying to correct the errors of the previous one. It starts with a simple model and iteratively adds trees to reduce residual errors from the previous trees. It is generally more accurate than a random forest, however it is more prone to overfitting if there is a lot of noise in the data.

Neural networks models are particularly good at capturing complex, nonlinear relationships between inputs and outputs. A neural network consists of an input layer, hidden layers, and an output layer. The input layer receives the features (e.g., program attributes derived from the AST), the hidden layers process these features using weights, biases, and activation functions, and the output layer provides the final prediction. Neural networks are highly flexible and can adapt to various problem domains, automatically learning feature representations without requiring extensive manual engineering. However, they require large datasets to avoid overfitting and significant computational resources for training.

Gaussian Processes are particularly interesting, because they take into account the probabilistic nature of energy measurement and provide a range of possible values alongside with the probabilities. This makes them ideal for tasks where understanding the uncertainty in predictions is important, such as energy modeling. However, their computational complexity grows significantly

with the size of the dataset, making them less practical for large-scale problems.

Support Vector Machines (SVMs) are a powerful tool in machine learning, capable of performing both classification and regression tasks. This algorithm identifies the optimal hyperplane in an N-dimensional space where it can separate all the features. When it is difficult to separate the features, a technique called kernel trick can be used to create an additional dimension to help separate them. This makes them good for high dimensional spaces, the ability to handle nonlinear relationships and the ability to ignore outliers. However, it can be harder to train this model, and tune the parameters.

Another alternative is to use genetic programming, which is not exactly a conventional ML algorithm, but rather a technique that can be applied to solve ML problems. It is a form of artificial intelligence inspired by the process of natural selection and evolution, where potential solutions to a problem are represented as programs or symbolic expressions. These programs improve iteratively, over generations, through mutations and crossovers, that sometimes can be random, guided by a fitness function. Genetic programming is useful for tasks like symbolic regression and feature classification. However, it can be computationally expensive and can produce inconsistent results due to its uncertain nature.

A good example of genetic programming in machine learning is Python Symbolic Regression (PySR). PySR builds on this technique to discover mathematical expressions that model data in a way that is both accurate and interpretable. Unlike black-box models, PySR generates human-readable formulas, making the results more interpretable and transparent. PySR works by searching the space of mathematical expressions to find those that best fit the data:

- It starts with a random population of simple mathematical expressions, such as $x + 1$ or $\sin(x)$.

- Evaluates how well each expression fits the data using a loss function (like MSE).

- Expressions are then evolved over several generations using operations like mutation and crossover to produce new, often more complex formulas.

At the end of the process, PySR provides a list of candidate expressions, typically sorted along a Pareto front balancing complexity and accuracy. While more complex expressions often provide better predictions, users can select simpler ones if interpretability is a priority.

This technique is particularly well-suited for energy prediction, as it can reveal easy-to-understand formulas that explain how certain parts of a program contribute to energy consumption. This can help developers understand and optimize the energy efficiency of their code more effectively than with traditional black-box models.

These algorithms were taken into account when developing the model that can best predict energy. Specifically, the models evaluated in this project include: Decision Tree Regressor, Random Forest, Gradient Boosting, Linear Regression, and PySR. These algorithms were chosen based on characteristics such as the simplicity of linear regression, the predictive strength of trees, and

the interpretability of symbolic regression with PySR. Together, these models offer a balance of accuracy, efficiency, and explainability, which is important for modeling energy consumption in Java applications.

Chapter 3

Related Work

This chapter discusses relevant approaches to energy measurement in software, its importance, and how to achieve it effectively. The chapter is divided into three categories: general context and approaches, static analysis-based tools, and dynamic analysis-based tools. This structure allows for a clearer comparison of technical strategies, their strengths, and their limitations.

3.1   General Context and Approaches

Energy efficiency is a critical focus across industries, as it directly impacts global sustainability, economic costs, and product quality. The goal is to reduce greenhouse gases to create a sustainable future, reduce infrastructure costs, and improve product quality [1].

In particular, large scale computation and communication consume a lot of global energy, and these values have been increasing in the last decades, so the topic of energy aware programming and energy efficient software has been targeted by many researchers in recent years with the objective of reducing energy costs in large IT infrastructure [30]. This improvement can be considered an optimization problem and can be tackled in several ways for example a heuristic approach by adjusting the hardware performance dynamically, or completing tasks in their deadlines, using the least energy possible. However, some of these implementations can only be short term solutions and in long term, the focus will be toward more complex models that can predict and optimize performance relative to hardware configurations [2].

An approach to increasing developer's awareness of the energy consumption of their code involves creating extensions to already used programming languages, such as Java. For example, ECO [31], a programming model as a minimal extension of Java. By rewriting some parts of the code to this extension syntax it is possible to define resource limits on the battery or temperature implementing adaptive behaviors through modes, and leveraging runtime monitoring.

In addition, new languages can be developed to address these goals, as demonstrated by ENT [32]. ENT is a Java extension that empowers programmers with more direct control over the energy consumption of their applications. ENT's type system enables applications to adapt dynamically to power constraints by switching operational modes based on resource availability, such as battery level or CPU temperature, allowing for software-level energy optimization. How-

ever, the language introduces complexity, making it potentially challenging for developers to learn and adapt to existing codebases.

Using machine learning algorithm has also shown to be effective to estimate energy consumption. Fu et al. [33] used four distinct ML algorithms (Ridge Regression, Linear Regression, Lasso, and Random Forest) to analyze the energy consumption of various apps, achieving low average error rate. These findings demonstrate the potential of such models to serve as the foundation for future tools, enabling developers to predict and optimize software energy usage without relying on specialized hardware. Other works go even further with using machine learning for energy efficiency and explore LLMs for generating not only optimized code but even reviewing and refactoring software for sustainability. Rani et al. [12] demonstrates that while LLMs recommend a range of optimizations, from vectorizing to memory management, their effectiveness in terms of actual energy savings remains low with LLMs usually making trade-offs between memory and execution time. Cappendijk et al. [34] investigate prompt-based optimizations by modifying LLM inputs with sustainability-focused instructions. While some prompt–model–task combinations achieved up to 59% reduction in energy consumption, the results were highly inconsistent, with some prompts even increasing energy usage by over 400%. Their findings underline both the potential and current limitations of using LLMs to directly influence software energy efficiency through prompt engineering.

Estrada et al. [35] proposed an energy consumption prediction model to optimize energy management in cloud and fog infrastructures, addressing challenges such as high operational costs and environmental impact. Their system integrates machine learning with sensor-based hardware to create a non-intrusive monitoring approach. Using a network of sensors to collect real-time data on metrics like voltage and power, processed via MQTT and visualized on dashboards, the study employed a robust linear regression model to predict hourly energy consumption. The research emphasizes the importance of real-time monitoring and machine learning integration for achieving energy efficiency in data centers, aligning with Green IT principles.

3.2   Static Analysis Tools

The use of static analysis can be valuable for understanding how instructions affect the energy consumption of programs. Aggarwal et al. [36] shows that system calls are directly related to energy consumption in Android applications. With this insight, it's possible to use static analysis to identify system calls within the code. This information can then be used to infer potential energy usage patterns, providing an early indication of where higher energy consumption may occur. This approach highlights the importance static analysis can have to understand program energy behaviors.

To tackle the problem of energy consumption in IT, some solutions have been presented. Some researchers focused on using energy measurement tools, like JRAPL to measure common libraries in Java and understand how much energy they use and what are the best alternatives to improve the energy efficiency of the code [37]. Observing common libraries for the implementation of list, sets

and maps, is possible to see which ones have the better energy efficiency and what changes could improve the code. Hasan et al.'s [38] research adds to this by creating detailed energy profiles for various Java collection classes, including lists, maps, and sets, across different implementations (Java Collections Framework, Apache Commons Collections, and Trove). Their work presents concrete quantification of energy consumption in these collections based on common operations such as insertion, iteration, and random access, and highlights the performance impact of collection types on energy efficiency for different input sizes.

However, because these collections are often used with threads, it is important to understand how much energy efficiency can be improved without compromising thread safety. The energy consumption of Java's thread-safe collections was studied by Pinto et al. [39], where researchers demonstrated that switching to more energy-efficient collection implementations can reduce energy usage while maintaining thread safety.

Building on these efforts, Pereira et al. [40] introduced a static analysis tool (Jstanley), as part of an Eclipse plugin, that can detect energy inefficient collections and recommend better alternatives. While Jstanley demonstrated notable improvements in energy efficiency within its specific context, it has several limitations. For example, they only account for three collections interfaces, (Lists, Sets and Maps) and only account for three different sizes (25,000, 250,000 and 1,000,000), it does not account for loops, thread safe and thread unsafe collections. Compared to our approach, Jstanley is limited, as it does not provide the actual information about the energy spent, it just shows recommendations. While the tool shows great improvements in its tested environment, replacing collections may not be enough in many practical cases. A more extensive tool, capable of analyzing a wider range of collections and providing energy metrics, would enable developers to achieve even greater energy efficiency and awareness.

Oliveira et al. [41], proposed a tool (CT+) that is capable of performing static analysis of the code, recommend changes that reduce energy consumption and automatically apply these changes. It is achieving up to 16.34% reduction in energy consumption even for a real-world, mature system such as Tomcat, it could reduce the energy consumption about 4.12%. It improved from previous works by taking into account more collections implementations, more operations, thread safety and support for mobile applications. Although CT+ is a significant improvement over previous tools, it still has some limitations. For example, it does not provide the actual energy consumption of the code, it just shows recommendations based on the previously collected statistics, as it did not leverage machine learning. Additionally, it only works with collections, and it is not easily generalizable for other context.

3.3   Dynamic Monitoring and Profiling Tools

In addition, SEEP [42] uses symbolic execution for energy profiling, generating multiple binaries representing different code paths and input scenarios. By analyzing these binaries with hardware-based energy measurement devices, SEEP provides energy consumption data, offering a deeper understanding of code efficiency across various inputs and paths. This approach complements

other tool, called PEEK [43], which builds on SEEP to help developers optimize energy usage with minimal effort. PEEK is an IDE-integrated framework that guides developers in writing energy-efficient code. It has a front end for IDE interfaces (e.g., Eclipse, Xcode), a middle end to manage data and versioning via Git, and a backend where energy analysis is performed, either through SEEP or hardware devices. Through these layers, PEEK identifies inefficiencies and suggests optimizations, supporting efficient coding practices. However, it has some limitations when compared to the proposed approach in this work, it uses dynamic analysis instead of static analysis, which was already explained in Section 2.4, why it was chosen over dynamic analysis. Additionally, it does not incorporate any machine learning techniques.

Some command tools, that work on Linux, help facilitate the process of energy measurement, like Perf [21], that is a command line tool already available in Linux, and is mainly used for performance monitoring and profiling. Although it's not specific for energy measurement, it can do it, with Intel RAPL but not as practical as other tools, specially when it's needed to measure a single process energy consumption. Powertop [22] is another tool capable of providing the power consumption, however it only works for laptops, as it requires to check the battery to see how much energy was used and calculate the power consumption.

JoularJx [23] is a Java agent that attaches to the Java Virtual Machine (JVM) at startup to monitor energy consumption. It runs in a separate thread, collecting CPU usage data for the JVM, its threads, and individual methods using statistical sampling. JoularJx performs power estimation by using platform-specific tools, such as the Intel API through the Linux RAPL interface, or a custom regression model for Raspberry Pi devices. It periodically analyzes stack traces to isolate energy consumption at the method level, taking into account execution paths and separating application-specific calls from system or agent-induced calls.

3.4   Limitations of Existing Approaches

The reviewed solutions showed strong potential in energy-aware software development, showcasing effective strategies such as including programming languages extensions, machine learning models, the use of static and dynamic analysis. However, these solutions have some limitations, like the need to execute the code before showing the average energy cost to the developer, having limited collections or lacking integration with the development workflow. These limitations point to the need for a more accessible, less limited solution.

The objective of this work aimed to develop a process that could obtain models that statically estimate program energy consumption, by combining established best practices with unique, domain-specific techniques. The final models produced by the framework pipeline can be trained using various language-specific collection libraries, and they are flexible enough to be modified or enhanced as needed. The final extension tool will then display the energy consumption of the code being analyzed making the developer more aware of the program's energy usage and helping him make better decisions.

Chapter 4

Approach and Implementation

The main goal of this work is to help developers understand the energy consumption of their programs, enabling them to make informed decisions about energy efficiency during development.

In order to achieve this goal, a framework [1] was built with the objective of simplifying and automating the process of benchmark generation, energy measurement, and model training. The framework is built with a modular approach, by making it easier to change in face of new needs. It is composed of four distinct modules, three implemented in Java and one as a standalone Python script, ensuring ease of installation and integration. Depending on the specific use case, these modules can function either as a unit or independently. Being the final component a simple and practical extension tool for VSCode, it can quickly and accurately estimate developers programs energy consumption. This allows them to get immediate feedback on energy consumption with every code change, facilitating energy-efficient development.

Many devices rely on Java and the JVM, so it is important that the code they run is energy efficient. Several factors can affect the power consumption of Java applications, including the behavior of the garbage collector and the efficiency of the memory management system [44] making it difficult to predict the power consumption of Java programs. This unpredictability highlights the need for a specialized tool to accurately measure and analyze power consumption so that developers can optimize their applications for energy efficiency. Java is an excellent choice for developing this framework because of its high interoperability with various operating systems and its widespread usage across the globe, making it a reliable and option. It has a wide range of useful libraries (JRAPL, JoularJx, Jalen) that help to measure energy accurately, and Java's typing and object-oriented features make the code easier to maintain and extend, so the tool can evolve with new energy metering standards and technologies.

There are several Java parsing tools available, such as WALA [45], SootUp [46], Spoon [47] and JavaParser [48]. WALA and SootUp are primarily designed for analyzing Java Bytecode and are generally more complex to use. For this project, Spoon was chosen because it is a user-friendly tool that facilitates easy retrieval and manipulation of the AST from Java source code. Both JavaParser and Spoon support AST manipulation and code generation. However, Spoon provides a deeper, type-aware metamodel and built-in templating features. These features make

---

[1]The full source code is available at: GitHub Repository.

Spoon more suitable for generating code that conforms to Java's syntactic and semantic rules, especially in complex transformation scenarios.
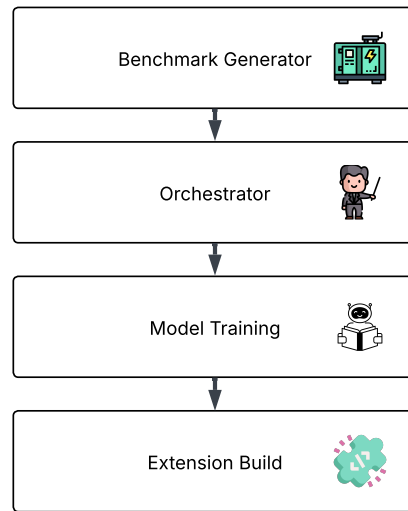
Figure 4.1: Main modules

To build the framework, it was necessary to design a system architecture capable of supporting the entire pipeline, from program benchmark generation and benchmark data collection to model training and prediction. The architecture is divided into distinct stages, each carefully analyzed and refined before progressing to the next. The architecture is divided in four main modules: **Benchmark Generator**, **Orchestrator**, **Parser**, and **Extension Tool**, as illustrated in Figure 4.1, will be explained in the following sections:

4.1 Stage 1: Benchmark Generator

To use machine learning models to predict energy consumption, it is necessary to have a large amount of data. This data can be obtained by generating benchmarks that are then executed to collect energy profiles. The benchmark generator is responsible for creating these benchmarks, which are then used to train the machine learning models. The generator is designed to be flexible and adaptable, allowing it to generate a wide variety of benchmarks based on different templates and input parameters. The architecture of the benchmark generator can be seen in Figure 4.2.

The benchmark generator works alongside with Java Spoon to make it the more general as possible allowing custom programs to be mass generated.

The generator is capable of generating benchmarks for custom, developer-created, Java classes, as well as for the most common APIs, such as collections interface implementations (Lists, Sets, Maps), and utility classes like Math, Base64, Duration. It can also be configured to analyze all public methods of a class or just specific ones.

```
1  import java.util.AraryList;
```

```
1  static class BenchmarkArgs {
2      public ArrayList<Integer> var0;
3      public Integer var1;
4      BenchmarkArgs() {
5          this.var0 = new ArrayList();
6          CollectionAux.insertRandomNumbers(var0,
7          1000,
8          "Integer");
9          this.var1 = 5;
10     }
11  }
12  BenchmarkArgs[] arr = new BenchmarkArgs[75_000];
```

Class/Collection

Template
- Inputs placeholder
- Type placeholder
- Array size placeholder

Input Tester
- Test inputs
- Test type
- Test array size

Save max input
- Max input
- Type
- Array size

Benchmark 1
Benchmark 2
Benchmark 3
Benchmark ...

```
1  in0 | arrSize: 75000 | type: Integer -> maxInput: 100000
2  in1 | arrSize: 75000 | type: Integer -> maxInput: 2584
```
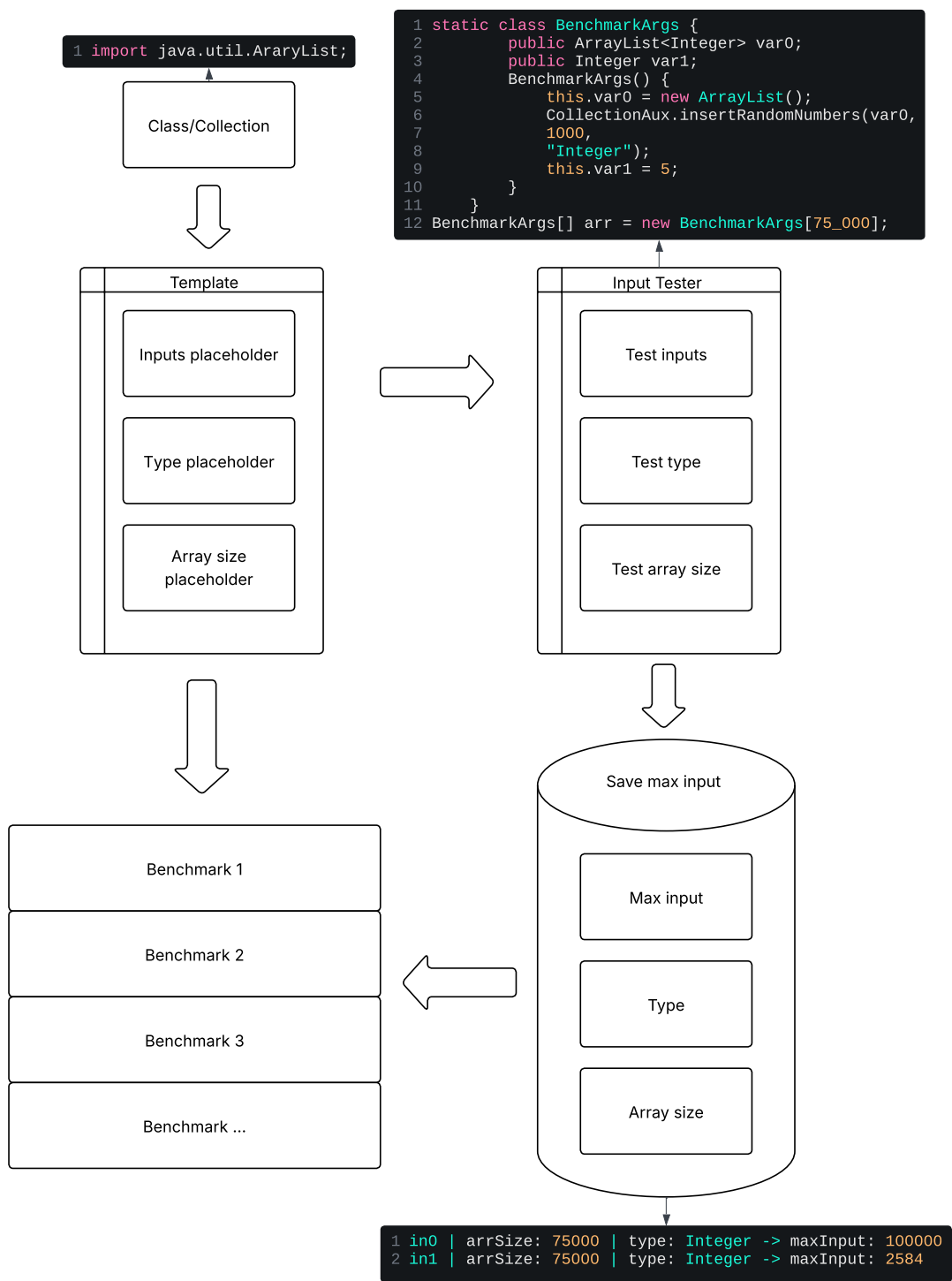
Figure 4.2: Benchmark Generator

4.1.1   Template Creation

The first step to generate multiple benchmarks is to first create an intermediate template capable of holding the necessary code that will later be used for energy profiling.

To generate benchmarks for a specific class, the user has to input the name of the collection or the name of the custom program it wants to generate. For the collections of Lists, Sets and Maps, the generator is prepared to create more than one implementation of those collections, for example, if the objective is to generate benchmarks to the List collection, the benchmark generator will use the `ArrayList`, `LinkedList`, `Vector` and `CopyOnWriteArrayList`. The user can also input the name of the method to be analyzed, or simply gather all the available methods in the class, which are then found using Spoon. After the search for the methods, the generator has access to all the methods it will analyze and their input parameters. Since it has access to the whole class, it can see how its constructors are called, and use them if any of the methods parameters requires. It recursively calls constructors if needed, making it very versatile to use. After identifying the methods that will be targeted, it starts by creating templates for each of them. The templates are Java classes that contain the necessary code to run the method, and placeholders for the inputs and types. The inputs we consider include the parameters received directly by the method, as well as the values required to construct any objects that are passed as parameters. The templates are structured so that they can be easily modified later, allowing the generator to create multiple benchmarks based on the same template. The templates are designed following the structure below:

- Input placeholders: Placeholders that will be later changed with real values for inputs, in this case inputs are variable values.

- Type placeholder: Placeholders that will later be replaced with Java wrapper classes.

- Array Size placeholder: Placeholder that will later be replaced with the value of an array size. (more details in 4.2).

Also, the template is structured so that the benchmarks will work in harmony with the **Orchestrator**, that will extract the energy profile for each benchmark (see Section 4.2), so when the placeholders are replaced with actual values, the orchestrator can run the program, and communicate with them.

Templates will have the definitions of a class that holds the necessary variables that the method under evaluation needs to run. Listing 2 shows an example of template to evaluate the method `List.add(Object)`. It follows the algorithm described in Algorithm 1. First it creates the list with the smallest constructor possible, then if the variable is a collection (List, Set, Map), or an array, it calls a custom-made method that populates collections with random values of a given type, and then it starts creating variables of parameters that the method `List.add(Object)` uses. The placeholder `ChangeValueHere1` will change to a random number, it contains a number `1` because it represents the input number of the method that will later help the model training

```java
static class BenchmarkArgs {
        public ArrayList<changetypehere> var0;

        public changetypehere var1;

        BenchmarkArgs() {
            this.var0 = new ArrayList();
            CollectionAux.insertRandomNumbers(var0,
            ↪ "ChangeValueHere1_changetypehere", "changetypehere");
            this.var1 = "ChangeValueHere2_changetypehere";
        }
    }
```

Listing 2: Example of variable placeholders creations

understand how inputs can affect energy consumption. The placeholder `changetypehere` later changes to a type. The template after the transformation can be seen in the Listing 3

It is worth mentioning that the types used by the generator are the Java wrapper classes, which are object representations of the primitive types. Using these types it is possible to achieve a more general generator, as every program can use them and if other custom types were used it would make the generator more complex and not general. Building on this idea, type placeholders are only generated for parameters that can accept multiple types. For example, if a method accepts a parameter of type `Object` or `ArrayList<T>`, the generator can substitute a variety of types, `Object` may become `Integer` or `Double`, and `ArrayList<T>` can be transformed as `ArrayList<Integer>`, `ArrayList<Double>`, or any other wrapper type. In contrast, if a method expects a fixed type like `int`, no type placeholder is generated, as introducing one would result in a semantic error due to incompatible types.

What mostly differs from template to template is the number of variables used, because different methods have different parameters, so the template can have more or less input placeholders, also the type placeholder changes according to the methods types and parameters.

Creating the template for each method allows cutting off time of the benchmark generation by only having to replace values of the placeholders instead of needing to create the whole benchmark all over again, since the benchmarks for the same method only differ in inputs, types and array size, maintaining all the structure.

4.1.2 Input Tester

An important aspect of the benchmark generator are the inputs it generates. Every method has its on funcionality that may be dependent on the input values. To be able to better generalize, it is important to find the method input values upper limit. Knowing the maximum size that different parameters can have is fundamental as it needs to be representative, so the energy profiles can

---

Algorithm 1 Template Creation Algorithm

    Given:
        $collection \coloneqq \{\text{List}\}$, collection selected.
        $ds \coloneqq \{\text{List, Map, Set, arrays}\}$, existent data structures.
        $methods \coloneqq \{\text{add, size, get}, \dots\}$, the set of methods of the classes.

  1: **procedure** CREATETEMPLATE
  2:      $methods \leftarrow$ GETMETHODS(`collection`)
  3:      **for all** method **in** methods **do**
  4:          vars $\leftarrow$ empty list
  5:          **for all** parameter **in** `method.parameters` **do**
  6:              $var \leftarrow$ PARAMETERCREATION(`parameter.type`)
  7:              **if** $parameter.type \in ds$ **then**
  8:                  FILLWITHRANDOMVALUES(var, maxSizePlaceHolder, typePlaceHolder)
  9:              **end if**
10:              Append var to vars
11:          **end for**
12:          CREATEBENCHMARKARRAYMETHOD(method, vars)
13:          CREATEBENCHMARKMETHOD(method, vars)
14:          CREATEORCHESTRATORSTARTSETUP
15:          CREATECOMPUTATIONMETHOD(method, vars)
16:          CREATEORCHESTRATORENDSETUP
17:          SAVETEMPLATE
18:      **end for**
19: **end procedure**
20: **procedure** PARAMETERCREATION(*type*)
21:      **if** ISPRIMITIVE(type) **then**
22:          **return** type
23:      **else**
24:          dependencies $\leftarrow$ GETCONSTRUCTORARGUMENTS(type)
25:          args $\leftarrow$ empty list
26:          **for all** dep **in** dependencies **do**
27:              arg $\leftarrow$ PARAMETERCREATION(dep)
28:              Append arg to args
29:          **end for**
30:          instance $\leftarrow$ INSTANTIATE(type, args) $\triangleright$ Create an instance, using the smallest
constructor.
31:          **return** instance
32:      **end if**
33: **end procedure**

```java
static class BenchmarkArgs {
        public ArrayList<Integer> var0;

        public Integer var1;

        BenchmarkArgs() {
            this.var0 = new ArrayList();
            CollectionAux.insertRandomNumbers(var0, 1000, "Integer");
            this.var1 = 10;
        }
    }
```

Listing 3: Example of variable placeholders replaced

cover more cases, but not to large so that the benchmarks start to run out of memory or taking too much time to complete. The limit definition works by using binary search. It has an inital threshold on the inputs (e.g., 1-100,000 to numerical values) and it starts by trying to run the benchmark with the half of the maximum upper limit threshold (e.g., 50,000).

It can be important, for pratical reasons, to impose a threashold on the execution time (in this project, we established the threshold as 10 seconds based on empirical experimentation). If the benchmark runs successfully, it will increase the input by half, and if it fails, it will decrease the input by half. This process continues until the maximum value for the input is found.

If the method that is being tested has more than one input, the input tester is responsible to find the upper limit for each parameter individually. First it sets all the input values to 1 and then starts the binary search individually for each of the parameters one by one while leaving the other parameters with the value of 1. Although this is a simplification, this method avoids having to find multiple combinations of parameters which would increase the time complexity exponentially. More robust solutions (such as using a combinatory search) could be implemented in the future. Nevertheless, even using the simplified solution, the process of finding the max inputs is time-consuming. To improve its performance, when the maximum value is found, the values of the input type, maximum value and order (e.g., first, second or third parameter), are stored in a file. This makes subsequent executions much faster. It is worthy to mention that the maximum inputs found depend heavily on the machine where the benchmark generation is taking place, as different hardware will change the maximum values allowed for the inputs. Example 1 illustrates how the input tester works for the method `List.add(index, Element)`.

> **Example 1: Input Testing Process for `List.add(index, Element)`**
>
> Consider analyzing the `add` method of a list with the following parameters:
>
> - `arg0`: Size of the list (integer)

- `arg1`: Index at which to insert the new value (integer)

- `arg2`: Value to be added (integer)

Step 1 – Varying `arg0` (list size):

- Iteration 1: `arg0 = 25,000, arg1 = 1, arg2 = 1`

- Iteration 2: `arg0 = 12,500, arg1 = 1, arg2 = 1`

- Iteration 3: `arg0 = 6,250, arg1 = 1, arg2 = 1`

- $\vdots$

- Final Iteration: `arg0 = 1,700, arg1 = 1, arg2 = 1`

Step 2 – Varying `arg1` (index):

- Iteration 1: `arg0 = 1, arg1 = 25,000, arg2 = 1`

- Iteration 2: `arg0 = 1, arg1 = 12,500, arg2 = 1`

- $\vdots$

- Final Iteration: `arg0 = 1, arg1 = 1, arg2 = 1`

Note: Since the list size (`arg0`) remains 1, the maximum valid index (`arg1`) is constrained to 1. This reveals a limitation in the input testing approach.

Step 3 – Varying `arg2` (value to be added):

- Iteration 1: `arg0 = 1, arg1 = 1, arg2 = 25,000`

- Iteration 2: `arg0 = 1, arg1 = 1, arg2 = 37,500`

- $\vdots$

- Final Iteration: `arg0 = 1, arg1 = 1, arg2 = 100,000`

Final stored input limits:

- `arg0 — integer — 1,700`

- `arg1 — integer — 1`

- `arg2 — integer — 100,000`

The fact that the input has a pre-determined range it allows using binary search which makes the search much faster than linear search. Nevertheless, the input search does not come without some limitations. First it is constrained to identifying valid input values within the range. As an example, throughout this project we dealt frequently with lists and collections, which require a minimum size of 1 to function correctly. Although this value can be changed in the future, for now it ensures compatibility with most common data structures. The higher bound was chosen due to

empirical evidences, as larger input values would lead to limitating execution times and memory crashes. Another constraint is on how the input handles multiple parameter methods. During its search for a valid input, it needs to fix all the other parameters that is not searching, with a default value (typically the lower bound). This approach simplifies the testing process and improves efficiency by avoiding the exponential complexity of testing all possible parameter combinations. However, it will introduce limitations in cases where the parameters are interdependent, which can lead to not estimating the real highest input possible. Despite these limitations, the input search plays a crucial role in ensuring that the benchmark generator produces viable test cases. By identifying input ranges that are both valid and computationally achievable, it reduces the unusable generated benchmarks (e.g., due to timeouts or crashes), and maximizing the number of generated benchmarks, that can be used for energy profiling.

4.1.3   Benchmark Generation

Finally, when the templates are created, and the maximum inputs are found the benchmark generation can finally begin. This part consists on picking every template created and replacing the placeholder values with actual values created by a random number generator. It follows the algorithm described in Algorithm 2

It starts by looping through the types and changing them to the Java wrapper classes, then it loops through the array size. Lastly, it loops through the input sizes determined by the input tester and can repeat this process a configurable number of times. By increasing the number of iterations, results in more benchmarks being generated with random inputs, constrained by the previously identified input bounds.

This process easily generate thousands of benchmarks which are crucial to train machine learn models that are able to predict energy consumption. When generating benchmarks, a number is chosen to balance the requirements for effective model training while minimizing the time spent on input testing and collecting energy profiles. Afterwards, that the benchmarks are ready to be compiled and used.

4.2   Stage 2: Orchestrator

Having all the benchmarks generated, it is possible to move to the next stage. Gathering the energy profiles for each of the generated benchmarks. Energy profiles [38, 41] are an established way to organize energy consumption data. They will be the main data source of the machine learning models to obtain accurate results. To reduce the complexity of this task, a process was implemented, illustrated in Figure 4.4. This process automates the task while taking into account what was mentioned in Section 2.1.

As described in the Section 2.3 there are several tools capable of performing dynamic energy measurements.

PowerJoular has good features that caught our attention, for example being a command line tool that could be easily integrated in almost every programming language, it stores the energy

---

Algorithm 2 Template Fullfillment Algorithm

Given:

$templates$, list of templates previously generated.

$types := \{$Integer, Double, Long, Float, Short, Character $\}$, types to be replaced.

$arraySize := \{75\,000,\ 100\,000,\ 150\,000\}$, array sizes to be used.

$template.maxInputs$, each template has a maximum input file associated to it.

$iterations$, number of iterations for the input loop.

1: **procedure** FULLFILLTEMPLATE
2:     **for all** template **in** templates **do**
3:         **for all** type **in** `types` **do**
4:             program ← REPLACE(template, "typePlaceHolder", type)
5:             **for all** arraySize **in** `arraySizes` **do**
6:                 program2 ← REPLACE(program, "arraySizePlaceHolder", arraySize)
7:                 **for** $i \leftarrow 0$ **to** `iterations` **do**
8:                     program3 ← REPLACEVALUES(program2, template.maxInputs)
9:                     SAVETOFILE(program3)
10:                **end for**
11:            **end for**
12:        **end for**
13:    **end for**
14: **end procedure**

15: **procedure** REPLACEVALUES(program, maxInputs)
16:     valuesToReplace ← FINDSTRINGSTOREPLACE(program, placeholderPattern)
17:     **for** $i \leftarrow 0$ **to** SIZE(valuesToReplace) - 1 **do**
18:         value ← valuesToReplace[i]
19:         type ← EXTRACTTYPE(value)
20:         max ← maxInputs[i]
21:         min ← MIN(1, max)
22:         newValue ← GETRANDOMVALUE(type, min, max)
23:         program ← REPLACE(program, value, newValue)
24:     **end for**
25:     **return** program
26: **end procedure**

---

used in a CSV file, capable of only reading energy of running processes and so on, as explained before in Section 2.3.

Since PowerJoular is a command line tool, it is launched as a process, and then it can be killed whenever needed because it is a process as well. This allows to measure not only programs/processes energies but have a more precise measurement, as it is possible to call PowerJoular to measure a specific computation and then kill it when the computation is finished.

With all of this in mind a process was built. The process uses an orchestrator that is responsible for invoking the target benchmark and the measurement tool (PowerJoular) to accurately measure the energy consumption of the benchmark or the specific computation being analyzed within it.

One challenge in measuring energy consumption is that computation often completes too quickly to capture accurately. It is hard to measure a single operation of, for example, a `List.add(Object)` with most tools, as it simply to fast for the tool to capture, and if the tool was able to capture it, the energy measured could have too much noise to be considered. Our solution was to loop through the method until the tool is capable of getting its energy and then dividing the total energy by the number of times it looped. This solution, although effective, introduces potential errors that need to be addressed.

By approaching the measurements with the loop technique, first we need to create the variables needed for the method target to analysis, like shown in the Listing 2

However, this raises problems with other methods, such as `List.add(Object)`. This method will insert values in the list, increasing its size. This increase in size can lead to different behavior of the method as the internal object changes in each iteration. In this case, inserting into a smaller list can produce different measurements of inserting into a larger list.

Now, suppose we have a method called `sort` that takes a randomly ordered list and sorts it. This introduces a problem when measuring its energy consumption. On the first run, the method will sort the unsorted list, which may take significant time depending on the randomness of the data and the sorting algorithm used. However, on subsequent runs with the same list, the input is already sorted, meaning the sort method will likely complete much faster or with minimal effort. This difference can result in misleading or inconsistent measurements.

To prevent this side effect from happening we designed a solution that consists of creating an array that holds the parameters, that can be seen in the Figure 4.3. The necessary parameters to execute the method are created and stored into the array. They are exactly copies with different refereces. Now, if one of the elements is changed during its execution on the method, it will not affect the other's execution. Since this approach increases memory usage due to multiple copies, we empirically selected array sizes (75,000, 100,000, and 150,000) to balance between avoiding out-of-memory errors and ensuring the computation runs long enough for PowerJoular to record energy measurements. PowerJoular by default needs to run for 1 second to create the CSV file that has the energy used for the target process, so these sizes aim so that looping the array takes at least a second. In some cases, the CSV file may not be generated because the loop executes too quickly. In such instances, the energy reading will be assumed to be 0 J, as the execution was too
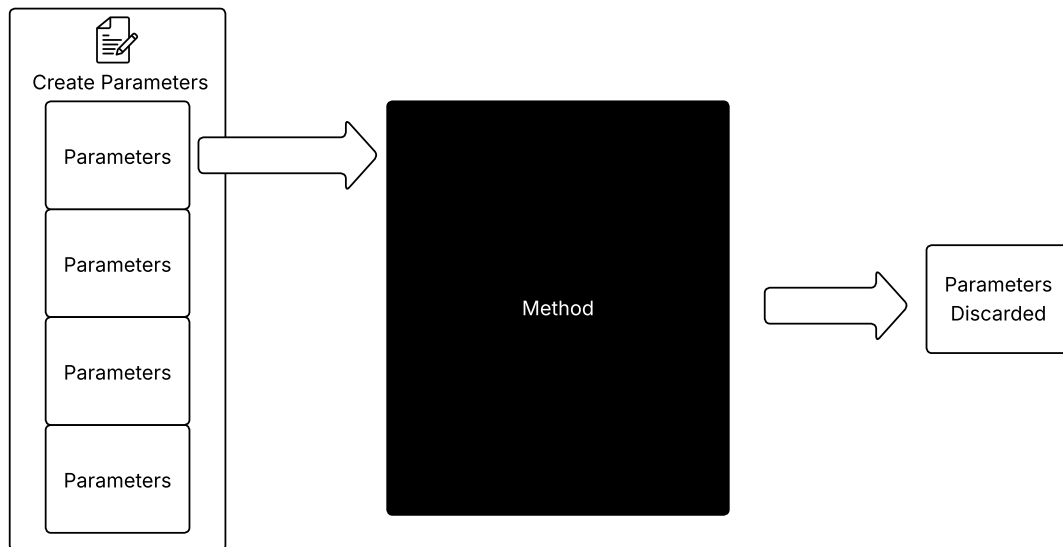
fast to record a value.

Figure 4.3: Array example

To be able to collect some faster executions, the original PowerJoular was modified to allow customizable measurement durations, enabling readings at intervals shorter than the default 1 second, such as 100ms, which provides greater flexibility and finer granularity in energy profiling.

The computation method shown in Listing 4 illustrates how each profiling method operates independently of the specific method being evaluated. It attempts to execute the target function repeatedly for approximately one second. In this particular case, the target function performs the operation `var.add(arg0);` and returns the number of iterations completed, which is then used for further calculations.

```
private static int computation(BenchmarkArgs[] args, int iter) {
        int i = 0;
        while (!TemplatesAux.stop && i < iter) {
                arrayList_add_java_lang_Object_(args[i].var0,
                ↪  args[i].var1);
                i++;
        }
        return iter;
    }
```

Listing 4: Computation method

With the process structure now defined, we can proceed to explain how it functions.

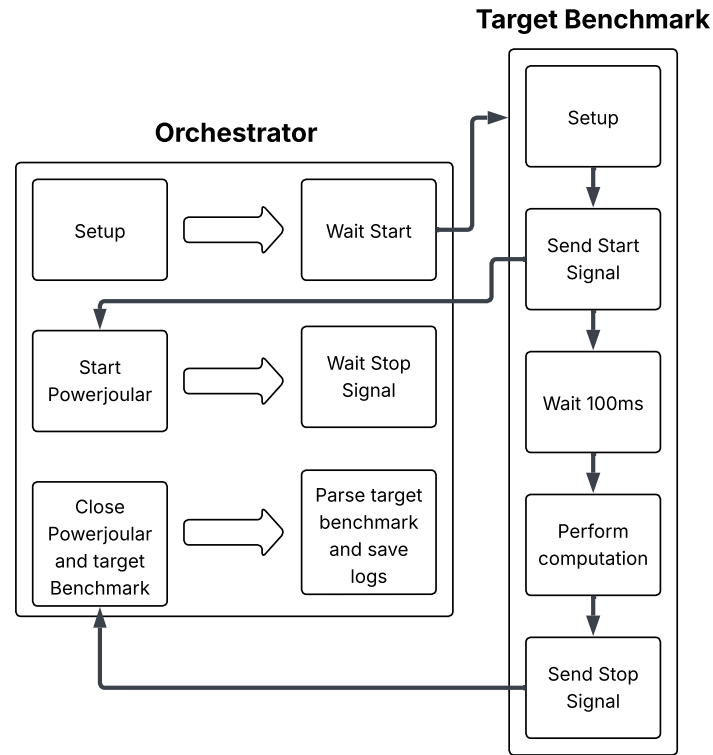The workflow of this step can be described as follows:

Figure 4.4: Orchestrator Workflow

- The orchestrator launches a command to start the target Java benchmark and waits a signal.

- The Java benchmark starts and setup the necessary elements to run (creating all the variables, reading/writing files, populating the array, etc.) and then before starting the computation it wants to measure, it sends a start signal to the orchestrator to start monitoring, and waits for 100 milliseconds.

- The orchestrator receives the start signal and reads the PID, which is stored in a file during the target benchmark setup. Finally, it starts PowerJoular using that PID. Then it waits for the stop signal.

- The Java benchmark will run until it finishes the computation. The computation runs for a maximum of one second. Then the number of iterations are stored in a file and the stop signal is sent back to the orchestrator.

- The orchestrator on receiving the stop signal, first stops PowerJoular and then stops the target benchmark, if needed. Then it parses the target benchmark to extract its features, combines them with the energy information stored in the files created by PowerJoular, stores it in a CSV file.

All these steps are performed for each generated benchmark. At the end of the process, a log

file is created containing key information, including all the benchmarks used, the PowerJoular files generated, temporary files, error logs, and features.

4.3   Stage 3: Model Training

First step to start training the machine learning models it is to extract features from the energy profiles. Features are the characteristics of the code that will be used to train the models and are based on the code structure and behavior. The features extracted can be seen in Table 4.1.

Table 4.1: Features extracted for Model Training, grouped by category

Feature Name	Description
Syntactic and Operator Counts	
`VariableDeclarations`	Number of variable declarations.
`Assignments`	Number of assignment operations (`=`).
`BinaryOperators`	Number of binary operators (e.g., `+`, `-`, `*`, `/`, `&&`).
`MethodInvocations`	Number of method calls.
`LiteralCount`	Number of literals (e.g., numbers, strings, boolean values).
Control Flow and Complexity	
`BranchCount`	Number of branching statements (`if`, `else if`, `else`).
`LoopCount`	Number of loops (`for`, `while`, `do-while`).
`LoopMaxDepth`	Maximum nesting depth of loops.
`CyclomaticComplexity`	The cyclomatic complexity of the code (number of independent execution paths).
Variable and Type Information	
`VariableCount`	Total number of variables.
`Reassignments`	Number of times variables are reassigned a new value.
`VariableTypes`	Types of variables used (e.g., `int`, `String`, `List<String>`).
`CustomMethodsUsed`	Number of user-defined (custom) methods used.
`CustomObjectsUsed`	Number of user-defined (custom) objects used.
`ImportsUsed`	List of imported Java packages or libraries.
`java.langMethodUsed`	If the method is from `java.lang`, the specific function name (e.g., `ArrayList.get`).
`Inputs`	The parameters received by the method.

There are a lot of possible ways of using machine learning, however in this case the approach that best fit our need is supervised machine learning. So, some models were trained to see how good they performed using the data collected.

On the context of this project, the first step it is to merge the features to start training the models. While each method is initially trained individually, it is not useful to treat `LinkedList.add(Object)` and `ArrayList.add(Object)` as entirely separate cases. Both represent the same `List.add(Object)` method, differing only in their characteristics specific to their implementation. Therefore, once all energy profiles have been collected, a merging step is performed to consolidate these related methods. All these merged features are stored in a new CSV and for the python program in a Data frame, to be processed.

To implement this functionality, Python scripts were used, combining libraries specialized on machine learning, like scikit-learn (sklearn) [49] which contains models and functions to evaluate the models, and PySR [50] (detailed in Section 2.6).

Among the available models, we selected a subset of them to train: Decision Tree Regressor, Random Forest, Gradient Boosting, Linear Regression, and PySR. Firstly the values of the MSE and R² are evaluated by the default values of sklearn. GridSearch was used to find the best parameters for a model. The GridSearch does not work with PySR as they are from different libraries so, the parameters for PySR were manually set. When the models are trained, they are moved to the extension folders ready to be used, and all the metrics for each model are logged to a file for subsequent analysis.

4.4   Stage 4: IDE Extension

The last component is the extension for an IDE, capable of predicting energy consumption. To provide this insight to developers it is necessary to build a tool that can provide all of that. Integrating the tool into an IDE makes it easier to understand and use, which are some of our main goals. With this the developer only needs to download an extension for an IDE and will access to the insights provided by the tool.

In this project, we choose the VSCode due to its lightweight architecture, extensive community support, and ease of extension development using familiar web technologies such as JavaScript and TypeScript. This extension allows the user to have a better understanding of how much energy the code is consuming, and what are the methods, and variables that most affect it.

It is important to note that the extension tool will serve as a guide, providing energy consumption estimates to raise awareness rather than dictate action. Ultimately, it is up to developers to decide whether to prioritize performance, energy efficiency or any other factor. For example, if a program only needs to run within a certain timeframe and can afford a slight reduction in performance, developers may choose to trade some performance for improved energy efficiency, making more informed decisions thanks to the insights provided by the tool.

To ensure efficiency, the tool uses static analysis to parse the code into an AST. From the AST, it analyzes the code and, using previously collected models, estimates the energy cost of the operations.

When opening the extension side page, it contains sliders that can change the input values, and it has the estimate button, to predict the energy.

The extension analyzes the user's Java files, identifies all the methods used, and matches them against a set of pre-trained models. Then it finds which variables affect those methods, meaning the variables that are the inputs to the method. For example, in `list.add(i)`, the relevant inputs and variables are `list` and `i`. The variable `list` is important because its size may impact energy consumption, while `i` is a method input whose different values can lead to varying method behavior, which can also impact the energy consumption. The extension does this to every method it finds in the source code and groups it by method. In the end it creates groups of input variables for each method, displaying it in sliders. The sliders allow the user to change the input values and when pressing the estimate button and, based on the values, it will change the energy estimation.

If a method calls another user-defined method that already has an assigned energy cost (but

was not used to train the machine learning models), then the calling method will also include that energy cost. This part was done by first analyzing individual methods for the model-trained methods and get their base energy. In a second iteration, each method is traversed to determine which calls are made, how many calls are made, and whether the calls are inside or outside loops. With this information, the total energy cost of each method could be calculated, accounting for both direct and indirect calls to model-trained methods.

When a model-trained method or a user-defined method are called inside a loop, a new slider appears to represent the loop size. The method's energy cost is multiplied by this loop size. In the case of nested loops, the energy cost is multiplied by the product of all nested loop sizes. Note that only methods and variables that affect the energy appear on the panel.

This extension can help understand how much energy the code is using. For example, the snippet in listing 5 counts how many times each word appears in a given text string, can be estimated for how much energy it uses.

```java
public class WordFrequency {
    public static HashMap<String, Integer> countWordFrequency(String
        text) {
        String[] words = text.toLowerCase().split("\\W+");
        HashMap<String, Integer> frequencyMap = new HashMap<>();
        for (String word : words) {
            if (word.isEmpty()) continue;
            String temp = frequencyMap.getOrDefault(word, 0) + 1;
            frequencyMap.put(word, temp);
        }
        return frequencyMap;
    }
    public static void main(String[] args) {
        String input = "Java is simple. Java is powerful.";
        HashMap<String, Integer> result = countWordFrequency(input);
        System.out.println(result);
    }
}
```

Listing 5: Java program to count word frequencies in a string

Figure 4.5 illustrates how the extension displays the energy consumption of the code in Listing 5. In the figure two methods countWordFrequency(String) and main(String[]) can be seen, next to their energy consumption estimations. Its also shown under the box with the method's name, the important variables that can affect energy consumption. For instance, in the method countWordFrequency(String) it can be seen that the size of the variables word and frequencyMap can change the energy usage. Also, the loop size is taken into account, and the number operations performed will impact the energy significantly.

When calling the method Map.put(Object, Object), the extension tool shows more

options available in the slider menu. The Figure 4.5, illustrates the variables that appear when
the method is used. It is noticeable that four sliders are available to change, and they contain
the three variables used in the method `frequencyMap.put(word, temp)`. The first one is
the collection input, `frequencyMap`, the second input is the String variable `word`, and the last
input is the String variable `temp`. All these three variables were detected by the extension tool
and marked as important. The last slider that is displayed in the image is the loop, which can also
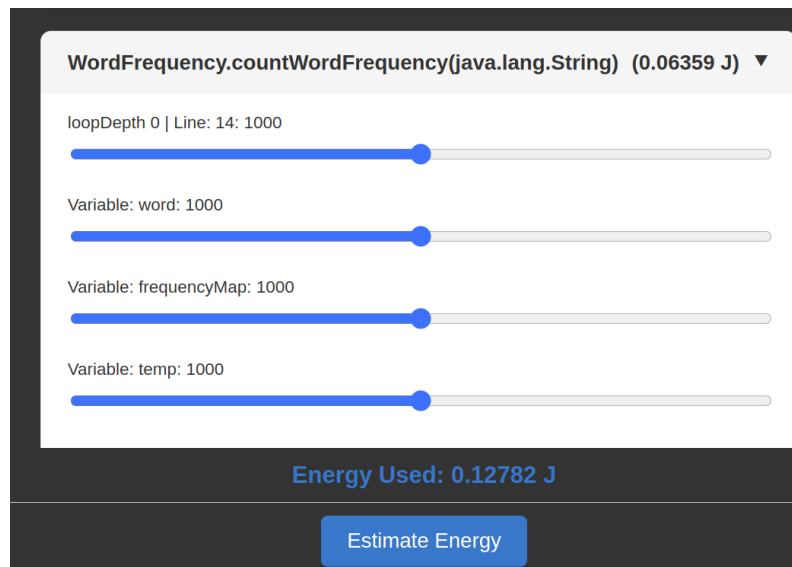affect the energy output, so it is also marked as important by the extension tool.

Figure 4.5: Extension example HashMap

Another example can be see in the Figure 4.6, where the same code is used, but instead of
using a `HashMap` it uses a `TreeMap`. The energy consumption is different, as the model was
trained to understand that the `TreeMap` consumes more energy than the `HashMap`.

There is also a feature that can help the user understand how the energy changes. When the
mouse hovers through some methods, it is possible to see the mathematical expression utilized for
the calculations. The Figure 4.7 shows how the expression is displayed in the extension UI. It has
the numbers that the model think are the best to predict the energy, and it has the features/variables
that it affects the prediction the most. In this case the two variables are the size of the map
collection and if there are any `TreeMap` collection being used. This can help understand what are
the variables that actually impact the energy of the code.

Another useful feature is the line that spends more energy in each method. Being the line
a user defined function or a model trained method, the line will appear highlighted, helping the
developer in case of needing help on where to start when needing to refactor the code for energy
efficiency. The example can be seen in the Figure 4.8.

The extension can be open in a Java project and gather the total energy used by the user made
methods, allowing for a better understanding of the code energy impact. Naturally, the extension is
limited by the set of pre-trained models it relies on. If a program uses methods that aren't covered
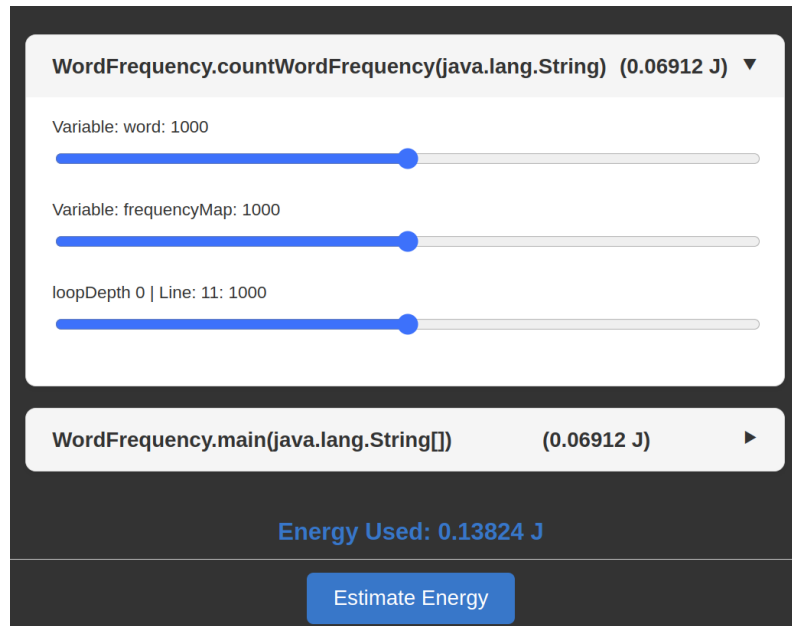
Figure 4.6: Extension example TreeMap

Figure 4.7: Expression for method Map.put(Object, Object)

Figure 4.8: Most expensive line highlighted

by these models, the extension will report an energy usage of zero, which is inaccurate, as those methods still consume energy.

This highlights the importance of continuously expanding the set of pre-trained models to cover a wider range of methods and scenarios. As more energy profiles are collected, the models will be able to predict more accurately the energy consumption of the code.

The extension is designed to be extensible, allowing for the addition of more energy profiles, features, and models as they become available. This ensures that the entire framework remains relevant and useful for developers as they work on increasingly complex projects.

Chapter 5

Results

In this section we will report the results of the experiments conducted to evaluate the framework and its components. The results will be presented in a structured manner, focusing on the performance of the benchmark generator, the orchestrator, the model training, and the extension tool. Each section will provide insights into the effectiveness of the framework in measuring and predicting energy consumption in Java programs.

Our hardware infrastructure is composed of two systems, one high-performance workstation used for data generation and collection, and a lower-spec machine used for testing the extension tool. The high-performance workstation is equipped with an AMD Ryzen Threadripper 3960X CPU, 94 GiB of RAM, and an NVIDIA GeForce RTX 3090 Ti GPU, while the lower-spec machine has an AMD Ryzen 5 3600 CPU, 16 GiB of RAM, and an NVIDIA GeForce RTX 3060 Ti GPU. The details of the hardware specifications can be found in Table 5.1.

Table 5.1: Hardware Infrastructure Specifications

Component	Specification
System 1 — Data Generation and Collection (High-Performance Workstation)	
OS	Ubuntu 24.04.2 LTS (x86_64)
CPU	AMD Ryzen Threadripper 3960X (24 cores / 48 threads). 2.2 GHz, up to 5.05 GHz
RAM	94 GiB
GPU	NVIDIA GeForce RTX 3090 Ti (GA102) with 24 GiB VRAM
System 2 — Machine for Testing (Lower-Spec)	
OS	Ubuntu 24.04.2 LTS (x86_64)
CPU	AMD Ryzen 5 3600 (6 cores / 12 threads). 3.6 GHz, up to 4.2 GHz
RAM	16 GiB
GPU	NVIDIA GeForce RTX 3060 Ti with 8 GiB VRAM

The more powerful system, System 1, as described in Table 5.1, was used for benchmark generation and data collection, as it was configured to work with SLURM (Simple Linux Utility for Resource Management). SLURM is a highly scalable, open-source job scheduler used to efficiently manage compute resources on shared systems. It allows tasks to be queued and scheduled based on resource availability and job priority, helping to organize workloads across users without manual intervention. The model training did not use SLURM, as it did not require significant

computation capabilities.

At the beginning of the experiment, some tools (described in Section 2.3) were tested to observe their behavior, understand how to use them, and determine which one best suited the needs of the project.

During the initial testing, Experiment Runner [25], a Python framework designed to facilitate experiment measurements was our tool of choice. The framework included an initial test template that used PowerJoular to measure energy of programs. While using the template and testing the framework some bugs and unexpected results were found, some of which were due to misuse of the framework. Due to these issues, a tailor-suited Java orchestrator was developed. Although it performed the same core function as Experiment Runner (measuring energy consumption), it was more straightforward, focusing exclusively on energy measurement rather than providing a general-purpose solution.

However, some discrepancies were observed between the energy values measured by the Java and Python frameworks. This was unexpected, as both used the same energy measurement tool (PowerJoular) and measured the same program in the same way. Still, the Python framework consistently reported lower energy values than the Java version. To investigate which tool was causing the inconsistency, three more orchestrators were implemented, including a simplified Python version. In total, four orchestrators were developed: Java, Python, C, and Bash. All four performed the same process, calling PowerJoular to measure the energy usage of a Java program. This setup represents an early iteration of the approach later detailed in Section 4.2, which utilized signal-based control to ensure that the measurement tool only ran during the exact computation period being measured.

Figure 5.1 presents the used CPU power observed by the four orchestrators while running 100 times a Fibonacci recursive program written in Java. They are ordered by the less energy to the highest energy.
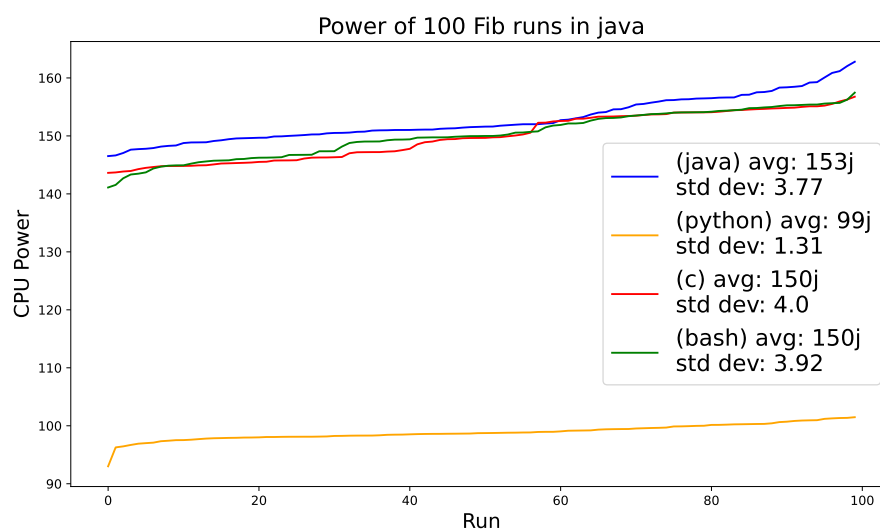

Figure 5.1: Orchestrators Comparison

It is noticeable that the Python orchestrator read energy values lower than the other orchestrators. Further analysis of the orchestrators revealed a notable difference in behavior. When the Python orchestrator was running, both the parent and child processes consumed CPU resources. In contrast, the other orchestrators (Java, C, and Bash) showed CPU usage only in the child process. This disparity may explain why PowerJoular reported lower energy consumption for the Python orchestrator. Since the CPU load was shared between the parent and child processes, PowerJoular, which measures energy only for the child process (in this example, the target Fibonacci program), captured less total energy usage. Since the experiment runner included an example demonstrating how to use the framework with PowerJoular, the authors were made aware of this potential conflict when launching PowerJoular from Python, having been warned of this potential conflict via email.

5.1   Models Metrics

To instantiate the tool, models were trained on methods from the Java collections framework, specifically focusing on the `List` and `Map` interfaces. Data for these methods was generated, collected, and used to train the models. With the models in place, it is now possible to evaluate their performance using metrics such as R^2 and Mean Squared Error (MSE), based on the features extracted during data collection.

Figure 5.2 presents the R^2 of all the methods that were analyzed from the `List` and `Map` interfaces. A score of 1 would mean that the model can get 100% of the prediction right.However, most of the time, it is not feasible to achieve a score of 1. In fact, most values are really low (bellow 0.5), which means that the model cannot predict the energy very well for some methods. The best models were for the method `addAll()` and `containsAll()` of the `List` collection, which got an R^2 of around 0.8 for most models.

These three methods with the highest accuracy were the ones that generated bigger energy outputs, because the methods were computationally more intensive than the others. This made it so that bigger inputs would result in bigger energy outputs, and make the models more easily find a pattern to predict the energy. As for the other models, since a bigger input would not mean a bigger energy output, the model, might have some difficulties predicting the energy. This does not mean that a lower R^2 will completely make the model unusable for some methods, as their MSE, that can be seen in the Figure 5.3 is not high, meaning that even when failing to predict the energy of the method, the failed prediction will not be far away as one might expect. For example, if a model has an MSE of 1×10^{-10} and predicts the energy usage to be 1 J, then even if the prediction is not exact, the true value is likely within $\pm\sqrt{1 \times 10^{-10}} = \pm 1 \times 10^{-5}$ J of the prediction, indicating accurate performance despite a potentially low R^2.

Another factor that may contribute to low R^2 scores is the behavior of the energy measurement tool when applied to certain methods. For instance, some methods, such as `size()`, do not require more computational effort as the collection size increases. Whether the collection has one element or one million, the method executes in roughly the same amount of time. Consequently, the energy consumption remains nearly constant, regardless of the input size. Since these methods
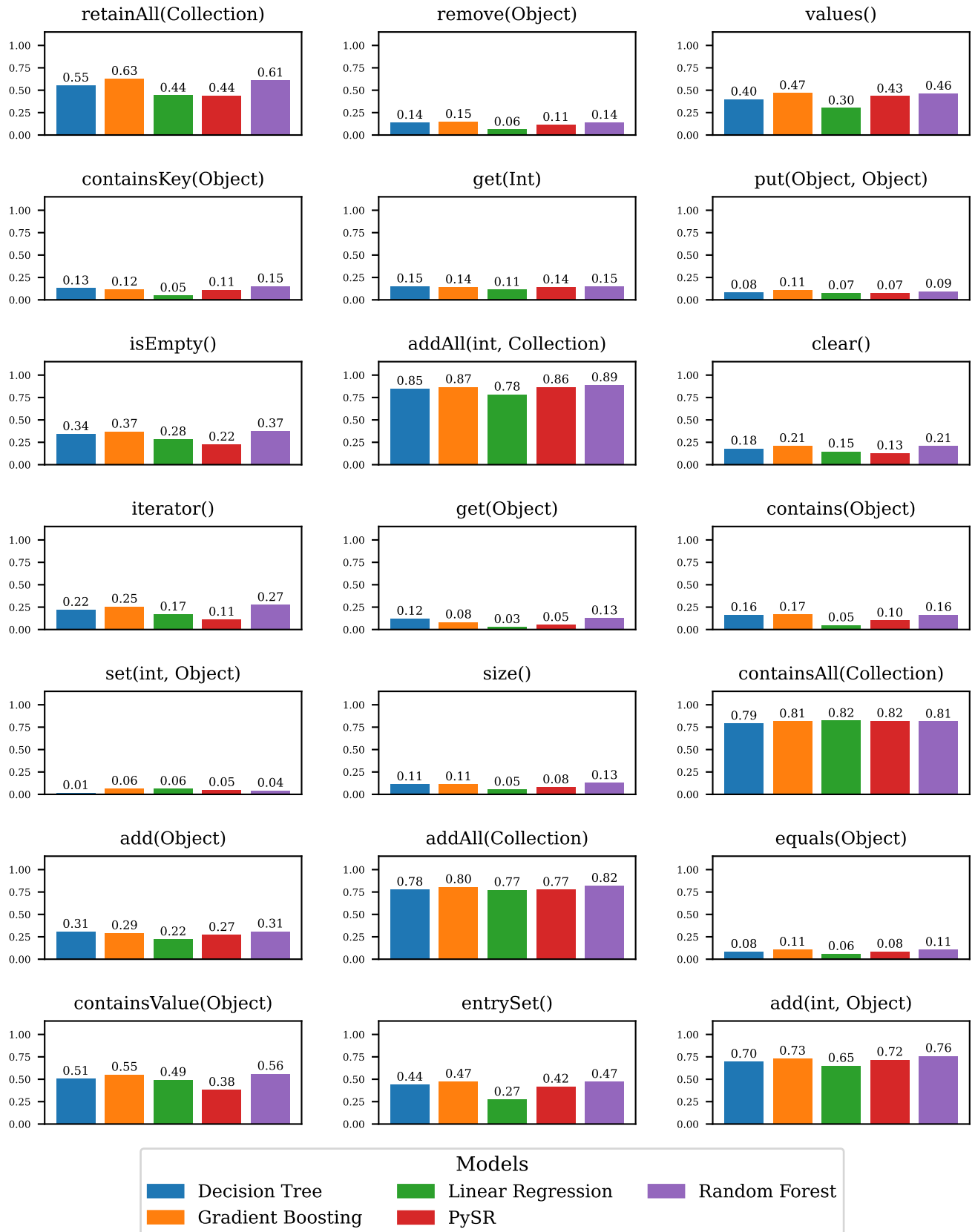
Figure 5.2: R² Comparison (*Higher is better*)
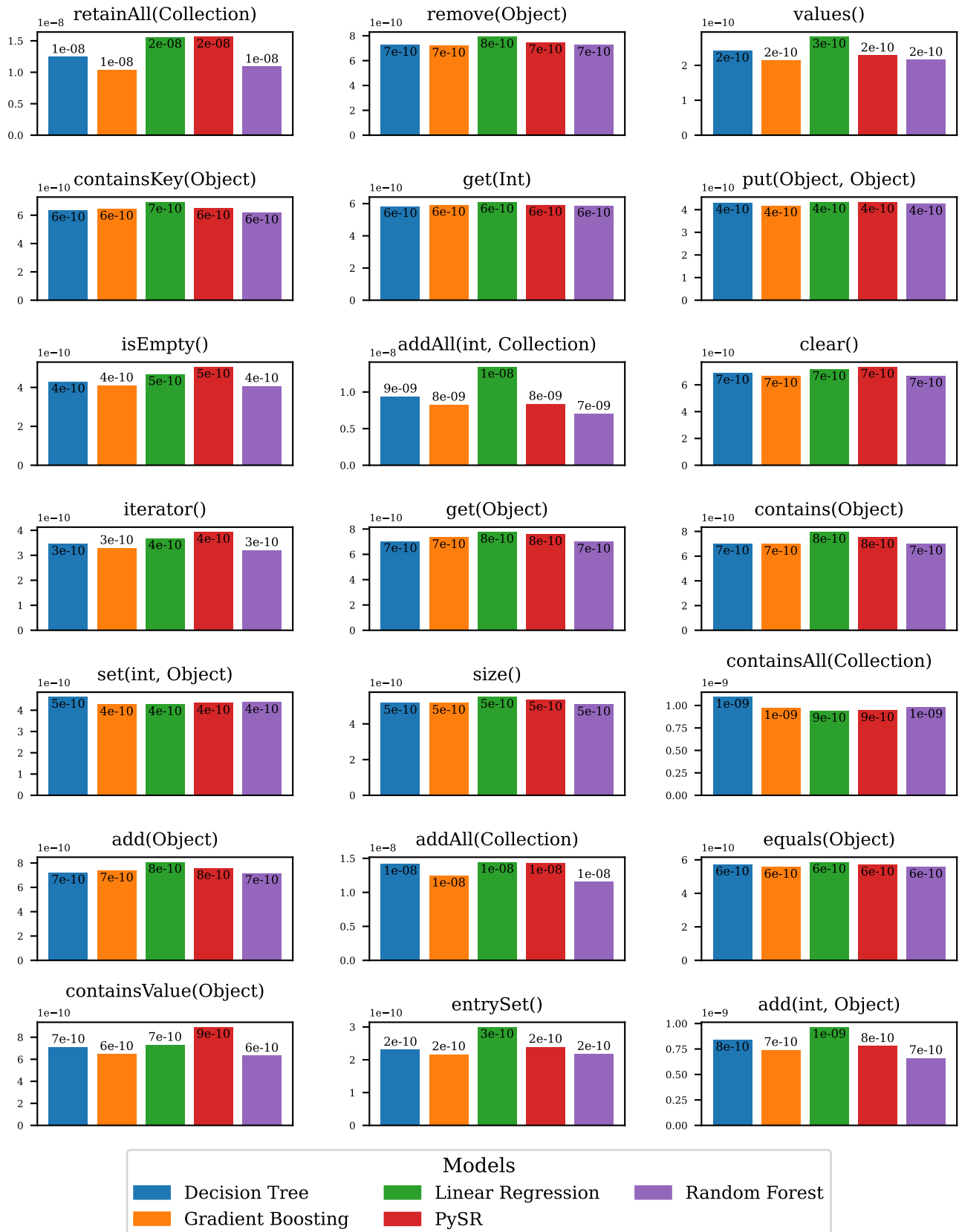
Figure 5.3: MSE Comparison (*Lower is better*)

complete quickly and consume little energy, even minimal measurement noise can significantly affect the recorded values. Figure 5.4 illustrates this effect: although an increase in energy with input size is typically expected, the recorded energy values for `List.size()` remain considerably flat. While this example isolates a single feature, and other features also influence energy consumption, input size is often the most significant. This observation suggests that for low-energy, low-variance methods, measurement noise can impact the signal, making accurate energy prediction particularly challenging.
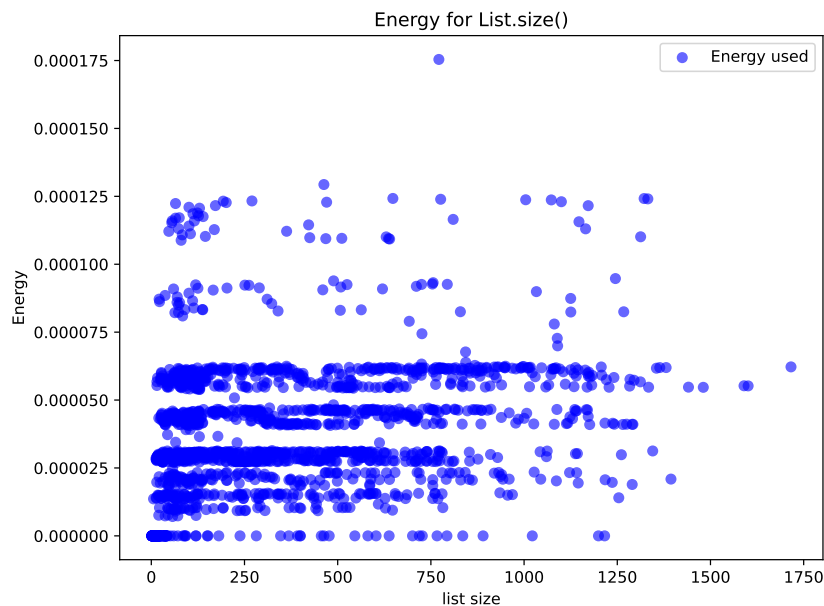

Figure 5.4: Energy for the `List.size()` method with different list sizes.

Nonetheless, the lower R² scores should be addressed, and what can be done to improve the results of these models is to have the benchmark generator create higher inputs, so the energy profiles also have outputs with higher energy, making the energy predictions more accurate. Then, since most of the predictions depend on the input, it means the other features do not have such higher impact, so it is also important to pick better features and remove others that might not be interesting. It is possible to improve a complex method that relies on a simpler method by incorporating the features or predictions of the simpler method, which can help improve the accuracy of the model for the complex method. In particular, accessing the energy profiles of simpler methods may refine the R² score of the complex method, as part of its behavior can be decomposed into more predictable components.

In general, most of the models present similar scores, however we used PySR in the instantiation. The biggest advantage over the other models is that it can represent the predictions in expressions which can help the users to try to understand why the code is using more energy. It has a nice feature of allowing to balance complexity and accuracy. And can easily be used in another code language as it is represented as a mathematical expression. The fact that most

models present similar results, ranging from advanced models like Random Forest and Gradient Boosting to simple models like Linear Regression, suggests that features used in the model likely do not capture highly nonlinear or complex relations. This indicates that energy consumption behavior in analyzed approaches can be reliably captured by simple relations. As a result, the set of features cannot offer the richness or variability needed for distinguishing more subtle energy behavior. This does not mean that the learning problem is simple, rather, it indicates that the available features may fail to express potential nonlinear patterns of energy consumption. Although the strategy should be enough for most cases, to improve prediction performance and enable more effective use of expressive models, future work should incorporate features capable of highlight the complexity of predicting energy usage.

5.2   Predicted vs. Measured Energy

Leveraging the developed models and the extenstion, we can present the results and evaluate how accurately the tool estimates energy consumption compared to real measurements obtained using PowerJoular. Since the energy profiles were generated on a specific hardware setup (see Table 5.1), the resulting models are adjusted to that particular system. If the tool were run on a different machine, the absolute energy values would likely differ due to variations in hardware characteristics. However, the relative differences in energy consumption between operations are expected to remain consistent. For instance, if `TreeMap` consumes more energy than `HashMap` on one system, the same trend will likely hold on another system, even if the exact energy values vary.

```java
ArrayList<Integer> l = new ArrayList<>();
ArrayList<Integer> l2 = new ArrayList<>();
l.addAll(l2);
```

Listing 6: Code example

To test the extension tool accuracy, the energy measurement needs to be run in the same machine were the data was collected. It is also important to take into account that the measured energies from these experiments were not processed exactly like the orchestrator, as the orchestrator processer has a more general approach that can introduce overhead in some cases. So the experiments performed, when possible, were only measuring the exact targeted method. For example, to measure the method `foo(type)` the orchestrator would need to create an array of parameters and loop through it during 1 second or until the end of the array size. However, during these experiments, if a single invocation of the method was sufficient for PowerJoular to capture its energy consumption, then only that single call was performed. This minimizes overhead and more accurately reflects real-world usage scenarios.

For this a simple program was developed, Listing 6, more like simple Java instructions, of just

creating two lists (both with size 1000) and using the method `addAll(Object)` and checking if the prediction matches the actual measurement.

Method	Actual Energy Range (J)	Predicted Energy (J)	Prediction Error (%)
`addAll(Object)`	4.9e-4 – 5.1e-4	5.51e-4	10.2
`addAll(Object) +` `size() +` `equals(Object)`	5.0e-4 – 5.2e-4	6.49e-4	27
`Map.put(Object,` `Object) + loop(1000)`	0.0027 – 0.0030	0.033	1058

Table 5.2: Comparison of actual and predicted energy consumption for different methods

The actual value measured by PowerJoular is in the range of 4.9e-4 to 5.1e-4J while the prediction is 5.51e-4J. This shows that the prediction (about 10.2% prediction error) was good for this method in particular. When trying to add two more operations (`size()` and `equals(Object)`) the measurement is in the range of 5.0e-4 to 5.2e-4J and the prediction is 6.49e-4J, about (about 27% prediction error). Usually, the method `addAll(Object)` is one of the methods that has the best accuracy of around 80%. Composing `addAll(Object)` with the other methods (`size()` and `equals(Object)`) that presented a low R^2 creates a more complex behavior and, as expected, increases the prediction error. Although the prediction is not as accurate as the previous one, it still provides a reasonable estimate of the energy consumption.

However, if we use the program presented in Listing 5, which involves a Map and a loop, the prediction accuracy drops significantly. The actual energy measured is in the range of 0.0027J to 0.0030J, while the prediction is 0.033J, resulting in a prediction error of about 1058%. This discrepancy highlights the limitations of the model when applied to more complex operations involving Maps and loops.

The models trained have a low accuracy for Map methods. The prediction error is very high, which means that the model is not able to predict the energy consumption of these methods accurately. This is likely due to the fact that the models were trained on a limited set of data and do not generalize well to more complex operations.

This issue also highlights a limitation of our instantation: by composing with methods that have low R^2 scores, the prediction accuracy is significantly affected. The models trained on methods with low R^2 scores struggle to accurately predict energy consumption for more complex operations, leading to high prediction errors, which leads to an error accumlation problem. Potential solutions could involve introducing a correction factor or calibration step based on empirical error measurements, adjusting the final predicted value to better reflect observed trends.

The results demonstrate that the extension tool is capable of providing useful estimations, especially in terms of relative trends between methods and input sizes. Proving it is capable of helping software developers on being more aware of the energy consumption of their codes. It shows potential in prediction, especially, for methods that are more impacted by the input. However, it is noteworthy that not all methods had the same high prediction accuracy as `addAll(Object)`,

and combining multiple methods or increasing program complexity tends to amplify prediction errors. These limitations highlight the importance of carefully selecting input sizes and operations for meaningful energy analysis, and they suggest that the tool is best suited for relative comparisons rather than precise energy estimation.

5.3   Evaluating Java Benchmark Programs

Our previous experiments were designed to evaluate simple methods, but we also used test cases considered more realistic, to further evaluate the framework's ability to analyze and collect energy models for various custom programs. The programs were selected from the Computer Language Benchmarks Game [51], which provides algorithmic benchmarks designed to test language performance on tasks that are representative of certain real-world computational workloads. The chosen programs were `BinaryTrees` (Java naot #3) `fannkuch-redux` (Java), `n-body` (Java naot #4), and `spectral-norm` (Java). These benchmarks were specifically selected because they are purely CPU and memory bound, with minimal variability, no reliance on disk I/O or multithreading, and no usage of external libraries. This makes them more predictable and reliable for energy consumption measurements. Some programs included several auxiliary methods, however, only those directly invoked by the `main` function were analyzed, as they were the most important and performed the core computations.

Starting with the program `BinaryTrees`, which implements a classic performance benchmark that builds, traverses, and destroys many binary trees to stress the memory allocation and garbage collection systems. It is composed of three main methods, some of which perform computationally intensive tasks, described as follows:

- `createTree:` Recursively builds a binary tree of the given depth. At each level, creates a node with two children (left and right), until depth equals zero. The result is a complete binary tree with $2^{depth} - 1$ nodes.

- `checkTree:` Recursively traverses the tree and counts the number of nodes. For leaf nodes, it returns 1. For internal nodes, it returns the sum of the recursive calls on the left and right subtrees plus one: $1 + \text{check(left)} + \text{check(right)}$.

- `trees:` Performs the core part of the benchmark by creating and checking many binary trees of varying depths and keeping one long-lived tree in memory to simulate memory pressure.

The `fannkuch-redux` program also presents a computationally intensive task. It generates all permutations of the sequence $[0, 1, \ldots, n-1]$ and, for each permutation, performs a "flip" operation until the first element is zero. A flip consists of reversing the order of the first $k+1$ elements, where k is the first element of the permutation. For each permutation, it tracks the number of flips needed (called the flip count) and maintains the maximum flip count across all permutations. Additionally, it calculates a checksum by adding or subtracting the flip count based

on the parity of the permutation index. The function returns the maximum number of flips needed for any permutation.

The `spectralnorm` program computes an approximation of the spectral norm (largest singular value) of a specific, implicitly defined matrix. It uses the power method, an algorithm that repeatedly multiplies a vector by the matrix to estimate how strongly the matrix can amplify a vector. The matrix A is not stored explicitly, instead, each element is computed on demand using the formula $A_{i,j} = \frac{1}{\frac{(i+j)(i+j+1)}{2}+i+1}$. The program initializes a unit vector and performs 20 steps of power iteration by repeatedly multiplying with A and its transpose A^T. The spectral norm is then approximated as $\sqrt{\frac{v^T(A^TA)v}{v^Tv}}$, where v is the resulting vector after the iterations.

The final program is the `nbody` program, which simulates the motion of celestial bodies under the influence of gravity, modeling a simplified version of the solar system. It tracks five bodies: the Sun, Jupiter, Saturn, Uranus, and Neptune. The simulation starts by calculating the total momentum of the system and adjusting the Sun's velocity to keep the center of mass stationary. Then, for a given number of time steps, it repeatedly updates the velocities and positions of the bodies using Newton's law of universal gravitation. Each iteration involves computing pairwise gravitational interactions between all bodies, updating their velocities based on the forces, and then advancing their positions. The program prints the total system energy before and after the simulation to verify that it remains nearly constant, which serves as a correctness check.

The Figure 5.5 shows the energy of each method, for different input sizes, for every benchmark generated and analyzed by the framework. It is possible to see that the methods have an exponential tendency with the increase of the input, which is expected from binary trees operations. It is also noticeable that the `checkTree` has a lower curve and maximum input, this was due to the fact that the method required that a tree was created before the method could be run, which would take more time, subsequently, the maximum input for this method was lower.

Table 5.3 presents the detailed information about the energy consumption of the BinaryTrees program. It is noticeable that the values predicted are not the most accurate, for some inputs it has good accuracy, for example, the method `createTree` has good accuracy for the input `26`. For other methods, such as `trees` it does not. Again as the previous experiment, the tool is not very accurate in terms of absolute values, and in this case for methods that change their energy considerably with minimum increase in the input, the models have a harder time to predict. The tool demonstrates good relative energy prediction, as input values increase, the predicted energy consumption also increases, and it decreases correspondingly when the inputs are reduced.

The Figure 5.6, shows the measured energy with the predictions, and it is observable that the energy grows quickly as the input also increases. Both the Table 5.4 and the Figure 5.6, confirm that for inputs lower than ten, the model has difficulties predicting.

For the `spectralnorm` benchmark, five methods are involved: `Approximate(int)`, `A(int, int)`, `MultiplyAv(int, double[], double[])`, `MultiplyAtv(int, double[], double[])`, and `MultiplyAtAv(int, double[], double[])`. While all these methods were included during the training, the analysis focused on `Approximate(int)`.
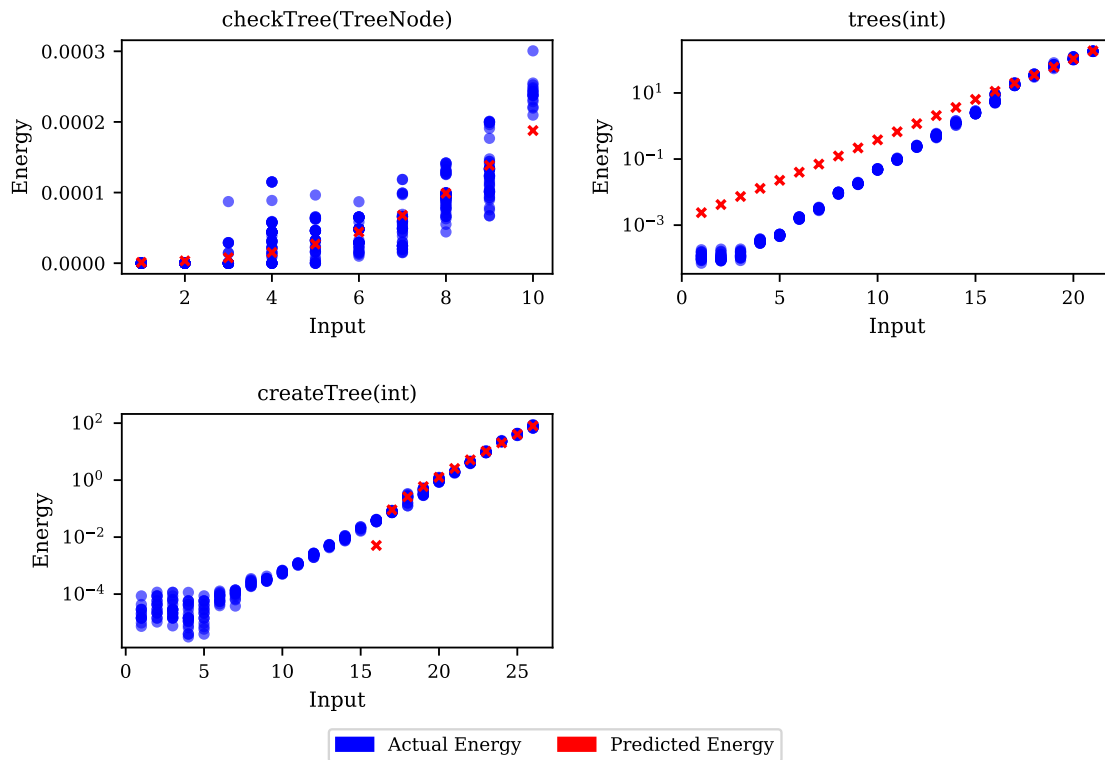
Figure 5.5: Energy for `BinaryTrees` methods

Method	Input Value	Predicted Energy (J)	Actual Energy Range (J)	Prediction Error (%)
`BinaryTrees.createTree(int)`	5	-0.082	[1.27e-5, 1.49e-5]	598070
	10	-0.083	[3.75e-4, 4.60e-4]	19980
	23	10.26	[11.78, 12.63]	15.9
	26	79.84	[77.29, 80.63]	1.12
`BinaryTrees.checkTree(TreeNode)`	5	2.71e-05	[2.63e-6, 2.70e-6]	916
	10	1.88e-4	[1.64e-4, 1.75e-4]	10.9
	23	2.19e-3	[1.37, 1.56]	99.85
	26	3.15e-3	[11.72, 13.29]	99.97
`BinaryTrees.trees(int)`	5	0.02	[2.68e-4, 3.55e-4]	7202
	10	0.38	[0.041, 0.045]	786
	23	581	[713, 800]	23.23
	26	3152	[6260, 7111]	52.86
`BinaryTrees.checkTree(TreeNode) + createTree(int) + trees(int)`	5	-0.06	[2.77e-4, 3.44e-4]	19341
	10	0.30	[4.59e-2, 4.77e-2]	541
	23	591	[796, 806]	26.20
	26	3232	[7062, 7350]	55

Table 5.3: Comparison of actual and predicted energy consumption for BinaryTrees program

This is because it is the method invoked by the `main` function and is responsible for the core computation of the program. The model obtained for the `spectralnorm` case prove quite effective discovering the pattern of the energy usage of the method, being the input the main feature that help achieve it, as Figure 5.6 shows, the prediction and measured values are almost similiar, and the prediction follows the curve of the mesaured energy. The concrete values can be seen in the Table 5.4, and it illustrates that the higher inputs tend to have higher accuracy as well.

For the `nbody` benchmark, three methods are involved: `offsetMomentum( double, double, double)`, `energy()`, `advance(double)`. The methods included in the training were `energy()`, `advance(double)`. However, the `energy()` method did not receive any inputs and lacked features that the model could use to differentiate energy consumption across executions. As a result, the model learned to predict a constant value for this method. Consequently, only `advance(double)` was selected for detailed analysis in this benchmark experiment. This time the resulting model had difficulties predicting the energy for the given method, as shown in the Table 5.4. The Figure 5.6 also shows an interesting behavior of the measured energy, which might explain why the model accuracy is so low.

Method	Input Value	Predicted Energy (J)	Actual Energy Range (J)	Prediction Error (%)
`NBodySystem.advance(double)`	100	4.4e-5	[3.02e-6, 3.25e-6]	1302
	1000	4.85e-5	[2.93e-6, 3.20e-6]	1480
	10,000	4.11e-5	[2.97e-6, 3.15e-6]	1243
	100,000	5e-5	[2.97e-6, 3.15e-6]	1533
`fannkuch(int)`	5	3.68e-3	[8.04e-5, 9.51e-5]	4089
	10	8.69	[3.72, 8.52]	42
	11	105.26	[87.25, 111.36]	6
	12	1275	[1187, 1585]	8
`spectralnorm().Approximate(int)`	10	6.29e-3	[2.65e-4, 2.85e-4]	2187
	100	.08	[0.02, 0.021]	297
	1000	2.67	[2.0, 2.22]	26
	10,000	212	[205, 207]	2.9

Table 5.4: Comparison of actual and predicted energy consumption for nBody, fannkuch and spectralnorm program

To achieve better results it could be possible to improve the feature's selection by incorporating characteristics that better capture the program complexity and increase the maximum input available for each benchmark generated in order to better represent the high-energy regions of execution.

This experiment also tests how pratical it is to actually measure a custom program. The `BinaryTrees` program, from how it is implemented on the website it needs some small adjustments in order to work with our framework. First, the program does not have a constructor for the class `TreeNode` which does not allow the program generator to populate the programs correctly, as the program on the website uses a custom method that create the nodes. Without this change, the methods that required `TreeNode` as input would be simply receiving an empty Object, as the constructor was also empty, making their generation useless as every program would be the same.
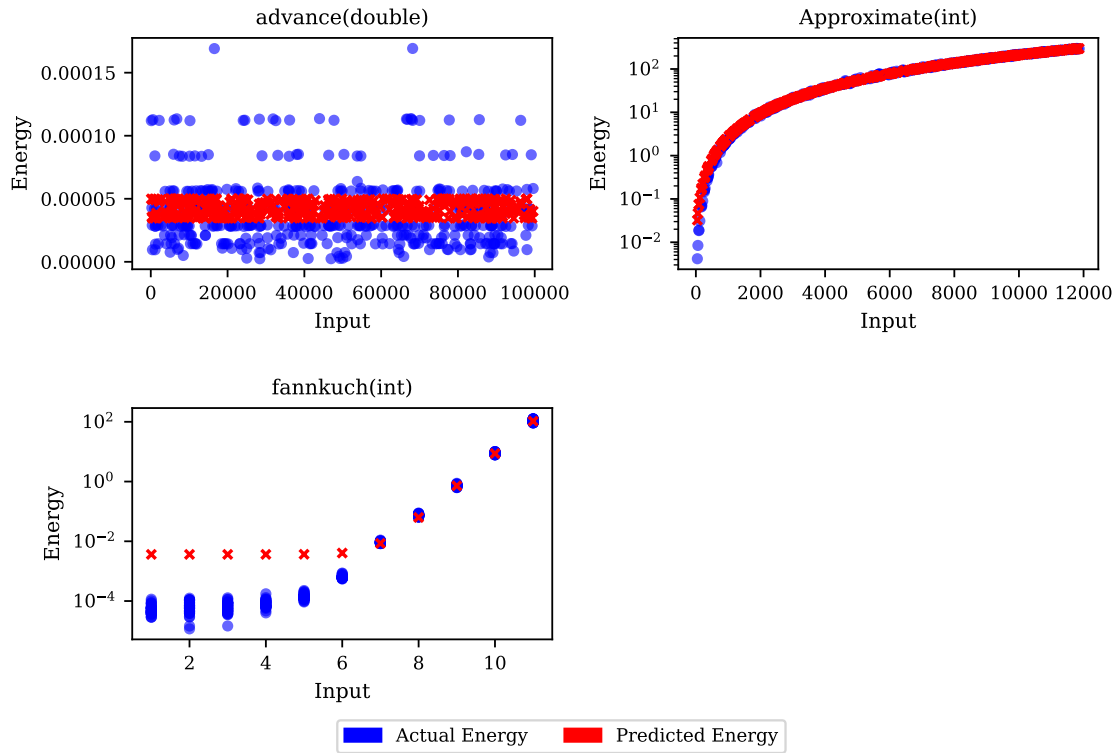
Figure 5.6: Energy for `advance`, `Approximate` and `fannkuch` methods

With that said, the only adjustments made to the class were, a new private empty constructor, that can only be accessed inside the class, so the generator does not detect it and use it, as the generator always tries to use the smallest one available, so it is important that the empty constructor is private. And it was added a new constructor, this one public, that had the same logic as the method that created the nodes. Now the methods that use `TreeNode` can be generated as the constructor can call the public constructor that has the similar behavior of the `createTree(int)` method. These changes do not impact how the program, or methods behave and only helps the benchmark generation. Aside from the `BinaryTrees` program, the others were even less troubled, as the `fannkuch-redux` and `spectral-norm` required no changes, apart from the package declaration, and the `n-body` program only required to divide the internal classes into more Java classes resulting in three files, `n-body.java`, `NBodySystem.java`, and `Body.java`.

This experiment objective's was not focused only on achieving the highest accuracy possible. It was also made to prove that it is possible to use the framework to test more interesting and complex cases. And it was proven that with minimal changes, it was possible to mass generate benchmarks, collect data and train models, for four different programs. While it may not work universally for all programs, since creating a framework general enough to handle every case is challenging, this experiment demonstrates that analyzing custom programs is possible.

5.4   Analysis

In most of the experiments, it was observed that the energy consumption curve followed the input values, meaning the model's predicted energy usage was largely dependent on the input. Since the tested programs involved computationally intensive tasks, increasing the input also led to longer execution times, which in turn caused higher energy consumption, indicating a strong correlation between execution time and energy usage. Table 5.5 shows the correlation between energy usage and input size. The Pearson correlation captures linear relationships between the two variables, which holds for most methods. However, for the `fannkuch` method, it shows only a moderate correlation, which does not reflect the observed behavior. By using Spearman correlation, which identifies monotonic relationships, the correlation becomes strong, aligning better with the results previously seen in Table 5.6.

Method	Pearson	Spearman	Interpretation
checkTree	0.83	0.90	Strong, slightly nonlinear relationship.
trees	0.84	0.97	Very strong nonlinear relationship.
createTree	0.54	0.97	Moderate linear relationship, but strong nonlinear.
advance	-0.05	-0.03	No meaningful correlation.
Approximate	0.97	0.99	Extremely strong, nearly perfect correlation.
fannkuch	0.52	0.97	Moderate linear correlation, but strong nonlinear.

Table 5.5: Correlation between energy consumption and input. Pearson measures linear correlation, while Spearman captures monotonic (nonlinear) trends.

For the methods where a lower input was enough to generate a benchmark that would run for an acceptable amount of time, the models could predict better, as a small increase in the input would affect the energy consumption by a significant amount. However, to prioritize accurate predictions for higher input values, many models tended to fail in predicting energy usage for the lower input range.

This trend may be attributed to the fast execution of the programs, when the input size is small, execution time is also short, which can lead to less accurate energy measurements. These quicker executions may result in underestimated energy consumption and introduce more noise into the readings. This could explain why the models struggle to make accurate predictions for lower input values but improve as the input size, and consequently, the program execution time and measurement accuracy, increases. Explaining why the prediction plots seen in the Figures 5.5 and 5.6, struggle to predict for the lower input values, and start getting better as they increase. For example, `fannkuch` with and input of 5 gets a prediction error of 4089%, and as the input increases the prediction error starts to decrease, for an input of 12 the prediction error is 8%. From the results, the best model obtain was from the spectralnorm `Approximate` method, with low prediction error values, followed by `fannkuch` method. While the `createTree` and `trees` methods demonstrate good performance, they are outperformed by the other two methods, showing higher prediction errors. All four models share a common trend, prediction accuracy improves with larger input sizes. The two methods that could not output satisfactory models were `checkTree`

and `advance`, which displayed the high prediction error values.

Both `createTree` and `trees` resulting expressions presented the exponential operation, aligning well with the trends observed in Figure 5.5. Specifically, the expression for `createTree` was

$$\exp\left((x - 19.571716) \cdot 0.6815255\right) - 0.08256852,$$

and for `trees`,

$$\exp\left((x - 11.710805) \cdot 0.563765\right).$$

However, `checkTree` did not have the same output. Its expression was

$$\left(x \cdot 1.7818553 \times 10^{-7}\right) \cdot \left(x^2 + 5.4003377\right).$$

From the plot, it appears likely that the energy output would continue to increase exponentially with larger input values. However, the input range was limited, since `checkTree` invoked `createTree`, which times out for higher inputs. As a result, the absence of an exponential component in the generated expression may explain its poorer prediction performance.

`Fannkuch` method also presented an exponential expression,

$$\exp\left((x - 9.133306) \cdot 2.4944384\right) + 0.003643311,$$

with better predictions for higher inputs. However, the data collected was limited to relatively small inputs, for example, an input of 12 would be impractical due to the factorial time complexity of the algorithm, which is $\mathcal{O}(n!)$. Therefore, to improve prediction quality in this experiment, especially for higher inputs, incorporating additional features could be beneficial.

`Approximate`, on the other hand, produced predictions that overall best matched the real values, the expression it generated used a sine function instead of an exponential or logarithmic form. That expression was

$$\sin\left(x \cdot 2.0593527 \times 10^{-6} + 0.0006085703\right) \cdot x.$$

As a result, for higher input values, the predicted energy output would begin to decrease and eventually become negative, an unrealistic outcome, due to the periodic nature of the sine function.

`Advance` was the least accurate method, and its resulting expression included a sine operation:

$$-7.739528 \times 10^{-6} \cdot \sin(x \cdot -0.81902814) + 4.2258347 \times 10^{-5}.$$

While this is not entirely incorrect when observing Figure 5.6, it suggests that the model was attempting to fit the data within a limited range. This behavior likely stems from the relatively poor quality of the energy measurements it received during training. Due to the absence of a clear trend and the narrow range of output values, the model defaulted to a bounded function like sine to minimize error. This results in predictions that visually resemble a flat band or rectangle in the center of the plot, as the sine function oscillates within an amplitude and fails to capture any meaningful global pattern.

From the experiments conducted, the one that displayed the most unusual result, was the `advance` method from the `nbody` program. The concrete reason to why the predictions and the measured values are not as accurate as the those of other methods, is not known for sure, however, there is a hypothesis. The methods `advance` required higher inputs. Since the input generator is limited to a maximum threshold, the benchmarks generated were also capped to that value, which would create benchmarks that would run for a small-time window. This would make that an input of 1 or 100,000 would spend in general the same amount of energy, making the input a bad decision factor for the model. Increasing the input could lead to a better model for this particular method, but could also increase the search time of the maximum inputs in the generator for other cases. As observed in previous results, the framework tends to struggles when the execution time of benchmarks is too fast, and can lead to inaccurate measurements. Interestingly, in the `advance` plot shown in Figure 5.6 a distinct pattern emerges, some readings form visible lines alongside the x-axis. This is unexpected, as short and noisy executions typically produce a scattered plot, similar to the bottom portion of the figure, where any input (x) can result in any energy output (y), however that is not the case. To better understand this anomaly, we further analyzed the data by categorizing it according to array size. In the Figure 5.7 the three different array sizes can be seen in distinct colors. The plot reveals that inputs associated with specific array sizes tend to align along horizontal lines, particularly at higher energy values, and the lower energy values are more scattered, which is what is expected in these noisy scenarios. The exact reason of this behavior remains unclear.
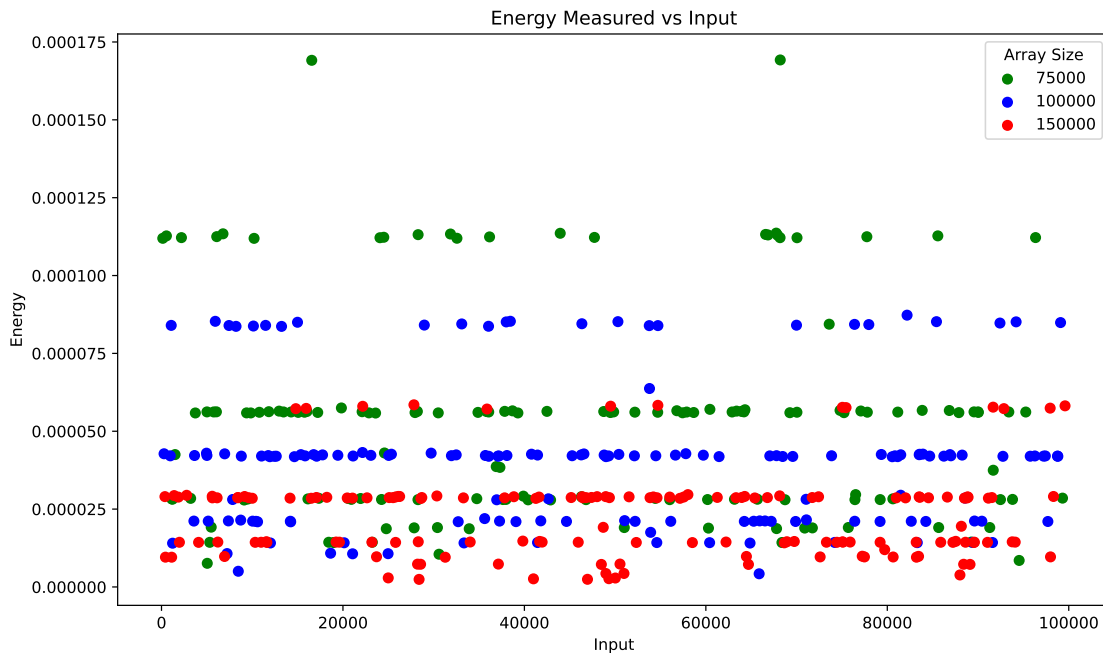

Figure 5.7: Energy for `advance` method divided by array size.

In its current form, the frameworks most important feature collected from the benchmarks is the input, and it showed to be effective in helping the models outputting a reliable prediction

model. However, it is possible to see that it is the only variable that can help the models predict in this case. They could benefit from a more diverse set of features that could include dynamic features, for instance, time of execution, processor cycles or other low-level metrics. Using more features could help the overall accuracy of the models.

Overall, the results indicate that while the models are capable of producing reliable energy consumption predictions under certain conditions, their accuracy and adaptability could be significantly improved with a richer and more diverse feature set. The conditions in which the framework displayed more potential was targetting programs that make use of inputs as it enriches the feature's set to reach a better model to estimate the energy consumption, and computive intensive tasks, which facilitate and improve the energy profiles collected by the framework, subsequently improving the resulting models.

5.5   Limitations and Challenges

During benchmark generation, most of the tested collections focused on `List`, `Set`, and `Map`. However, benchmarks were also generated for another collection category, the `Math` library. This introduced an issue. The computations in these benchmarks were extremely fast, completing too quickly for PowerJoular to measure any meaningful energy consumption. As a result, all energy readings for the `Math` benchmarks were reported as 0J. We were able to identify that the source of the problem was the array sizes used to hold the method parameters. While the three predefined array sizes worked well for `List`, `Set`, and `Map`, they were too small for the `Math` library. The smaller input sizes caused the `Math` computations to execute very fast, not giving PowerJoular enough time to capture energy usage. This highlights a challenge in energy profiling: some collections, especially those involving lightweight or highly optimized operations like `Math` functions, may be difficult to measure accurately. One possible solution would be to determine array sizes during benchmark generation, large enough to ensure that each benchmark runs for at least one second, giving PowerJoular sufficient time to measure energy consumption. In its current form, there are three array sizes as they were initially selected for Java data structures, but for other collections, like `Math`, these sizes do not fit as well, so it would be better to have dynamic sizes depending on the benchamark it is running. However, implementing this would significantly increase the time required for benchmark generation, as it would involve exploring many more combinations of input sizes to find suitable configurations.

Another important factor is that the tool is not able to predict energy when threads are involved, as the benchmarks generated only used a single thread, subsequently the models will not have that factor into account. This limitation exists because measuring and modeling the energy usage of multithreaded programs is particularly challenging due to factors like concurrent execution, thread scheduling, and synchronization overhead. However, threading can deeply impact the energy usage of a program, as a program execution stop being strictly linear, and can have multiple simultaneous computations, potentially increasing the code energy consumption.

A limitation detected during the experimentation of analyzing an external program, was how

the inputs on some programs are defined. For example, a program can have a method that receives a `String`, however, when inside the method it decides to convert the `String` to `int` and use it as an `Integer`, making it so a method that actually uses `Integers` and not `Strings`. Although it is easily noticeable by human eye, the benchmark generator cannot understand this kind of context changes as easy, and it would start generating inputs for `Strings` instead of `Integers`, which would fail in the benchmark generation. A good example of this usage is when the main function is called and receives the arguments in the `String[]` and then it uses its values in different variables.

Chapter 6

Conclusion

The use and management of energy has become a global issue, and the energy consumed by information and communications technology has a significant impact at a global scale. When developing programs, developers can have difficulties on how to improve the energy efficiency of their programs, due to the complexity and limited accessibility of relevant information as well as the lack of supporting tools and guidelines. This thesis reviews and builds upon prior research, tools, and techniques, by proposing an improved framework for estimating energy consumption. The framework integrates machine learning with static analysis to generate predictive models capable of estimating the energy usage of software programs. The framework presented is divided in four modules, and it eases the process of obtaining the estimation models, by generating the necessary data, collecting the energy profiles, training the dataset and presenting the results. By simply providing a program or selecting a Java collection, the framework can generate estimation models adapted to the input.

For developers who prefer not to wait for the full analysis process or are less familiar with energy-aware programming, our VSCode extension offers a quick and accessible alternative. By using the pre-trained models the tool provides immediate feedback on the estimated energy consumption of their code, without the need to execute it, helping raise awareness of energy efficiency in software development.

During the experiments, the framework proved it was capable of evaluating custom programs, with minimal modifications, which makes it reliable to obtain more data from across a wider variety of codebases. Additionally, the extension tool also proved it can raise energy awareness by using the trained models and static analysis, to provide real-time feedback on the energy impact of code changes.

In summary, this thesis contributes to the field of energy-aware programming by combining machine learning and static analysis to enable accurate and accessible energy prediction. Nonetheless, continued research and development are essential to further improve model accuracy, expand feature coverage, and support a broader range of programming scenarios.

Chapter 7

Future Work

The framework presented showed different modules that when combined result in a static analysis tool that estimates energy consumption of Java programs. Each of the modules can have individual improvements that can boost the overall performance of the final extension tool.

7.1   Enhancing Benchmark Generation

Some improvements can be performed to the Benchmark Generator. The first one is the max input finder. As it is now, the maximum input finder is a simple approach, as it was already explained in Section 4.1.2. This approach makes it, so it does not find the best combination of inputs available for each method, which can lead to the collected data to avoid edge cases. It is possible to implement a more robust solution that avoids increasing the time complexity by too much, and improves the searching capabilities of the input finder.

7.2   Improving Model Accuracy

In Section 5, some experiments were conducted and the R^2 and MSE metrics of the models were presented. It is observable that the overall accuracy of the models is not the greatest and that they can definitely benefit from a richer feature set. Estimating energy consumption is a complex process and requires multiple variables to actually get an accurate prediction, and the features uses at the moment might not be the ones the can better help the models achieve the best performance. During the experiment it was noticeable that the most important feature is the input size and the other features are not used as much. The models could benefit from more runtime features like time of execution, processor cycles, caches accesses and other low-level performance metrics.

7.3   Centralized Energy Estimation Model

The final approach uses individually trained models for each method, with each model tailored to its own features and prediction style. But when the dataset grows large enough, there is an opportunity to train a single, more powerful model that can directly estimate methods energy consumption, without needing to run the entire framework every time. By bringing together the

features, predictions, and real energy usage data from all the existing models, this unified model gains access to a richer dataset. That added depth allows it to make accurate energy predictions on its own. Each time the framework processes a method, the resulting model is not just pushed to the extension tool, it also helps improve the centralized model, by making it smarter and more accurate with every update. While the framework will still handle benchmark generation, energy measurement, and model training for the tool, over time we can consolidate these models towards one powerful model, eliminating the need to run the full pipeline for every new prediction, requiring only parsing the data of the target program and combining with the existing data. This model could update the extension tool with new data very quickly with little effort, however to achieve this kind of large dataset it is required a lot of time.

7.4 System-Specific Model Adaptation

There is an important aspect with the resulting models from the framework, they are all bounded to the machine in which they were trained. A model, for a given input, will always give the same output, despite the machine it is running. However, the process of collecting the model is repeated for different machines, the obtained models might differ as well. One useful feature that would make the models obtained more reliable would be to add the hardware information during the collection process. When there is enough data about hardware, the models can adjust the energy usage of the specific machine without having to collect the data all over again. This would be the most pratical for developers that would only want to use the VSCode extension tool with the existing models, without having the extra effort or time-consuming task.

Bibliography

[1] Kenneth Gillingham, Richard G. Newell, and Karen Palmer. Energy efficiency economics and policy. *Annual Review of Resource Economics*, 1(Volume 1, 2009):597–620, 2009.

[2] David J. Brown and Charles Reams. Toward energy-efficient computing. *Commun. ACM*, 53(3):50–58, March 2010.

[3] Google Support. Adaptive battery. `https://support.google.com/pixelphone/answer/7015477?hl=en`. Accessed: 2025-01-02.

[4] Google Support. Use battery saver on android. `https://support.google.com/android/answer/7664692?hl=en`. Accessed: 2025-01-02.

[5] Apple Support. Use clean energy charging on your iphone. `https://support.apple.com/en-us/108068#:~:text=With%20iPhone%2015%20models%20and,Clean%20Energy%20Charging%20is%20on`. Accessed: 2025-01-02.

[6] Android Developers. Power profiler for android studio. `https://developer.android.com/studio/profile/power-profiler`. Accessed: 2025-01-02.

[7] Anders Andrae. New perspectives on internet electricity use in 2030. *Engineering and Applied Science Letters*, 3:19–31, 06 2020.

[8] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[9] Constantinos A Balaras, John Lelekis, Elena G Dascalaki, and Dimitris Atsidaftis. High performance data centers and energy efficiency potential in greece. *Procedia environmental sciences*, 38:107–114, 2017.

[10] Gustavo Pinto and Fernando Castor. Energy efficiency: a new concern for application software developers. *Commun. ACM*, 60(12):68–75, November 2017.

[11] Tina Vartziotis, Maximilian Schmidt, George Dasoulas, Ippolyti Dellatolas, Stefano Attademo, Viet Dung Le, Anke Wiechmann, Tim Hoffmann, Michael Keckeisen, and Sotirios Kotsopoulos. Carbon footprint evaluation of code generation through llm as a service. In *International Stuttgart Symposium*, pages 230–241. Springer, 2024.

[12] Pooja Rani, Jan-Andrea Bard, June Sallou, Alexander Boll, Timo Kehrer, and Alberto Bacchelli. Can we make code green? understanding trade-offs in llms vs. human code optimizations. *arXiv preprint arXiv:2503.20126*, 2025.

[13] Md Arman Islam, Devi Varaprasad Jonnala, Ritika Rekhi, Pratik Pokharel, Siddharth Cilamkoti, Asif Imran, Tevfik Kosar, and Bekir Turkkan. Evaluating the energy-efficiency of the code generated by llms. *arXiv preprint arXiv:2505.20324*, 2025.

[14] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 22–31, New York, NY, USA, 2014. Association for Computing Machinery.

[15] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, page 57–76, New York, NY, USA, 2007. Association for Computing Machinery.

[16] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, October 2006.

[17] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Daniel Hackenberg, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt, and Wolfgang E Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204. IEEE, 2013.

[19] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, 2009.

[20] Intel Corporation. Running Average Power Limit (RAPL) Energy Reporting, 2025. Retrieved January 5, 2025, from https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[21] PerfWiki contributors. PerfWiki: Main Page, 2024. Last edited on 1 December 2024. Retrieved December 2, 2024 from https://perfwiki.github.io/main/.

[22] Arch Linux Wiki contributors. Powertop, 2024. Last edited on 26 April 2024, at 18:46. Retrieved December 2, 2024 from https://wiki.archlinux.org/title/Powertop.

[23] Adel Noureddine. Powerjoular and joularjx: Multi-platform software power monitoring tools. In *18th International Conference on Intelligent Environments (IE2022)*, Biarritz, France, Jun 2022.

[24] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[25] S2 Group. Experiment Runner. Retrieved November 15, 2024 from https://github.com/s2-group/experiment-runner/.

[26] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[27] Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. Editing support for software languages: implementation practices in language server protocols. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, page 232–243, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Microsoft. Language server protocol overview. https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/, 2025. Retrieved July 30, 2025 from https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/.

[29] Iqbal H Sarker. Machine learning: Algorithms, real-world applications and research directions. *SN computer science*, 2(3):160, 2021.

[30] Alcides Fonseca, Rick Kazman, and Patricia Lago. A manifesto for energy-aware software. *IEEE Software*, 36(6):79–82, 2019.

[31] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. A programming model for sustainable software. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 767–777, 2015.

[32] Anthony Canino and Yu David Liu. Proactive and adaptive energy-aware programming with mixed typechecking. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 217–232, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Cuijiao Fu, Depei Qian, and Zhongzhi Luan. Estimating software energy consumption with machine learning approach by software performance feature. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 490–496, 2018.

[34] Tom Cappendijk, Pepijn de Reus, and Ana Oprescu. Generating energy-efficient code with llms. *arXiv preprint arXiv:2411.10599*, 2024.

[35] Rebeca Estrada, Danny Torres, Adrian Bazurto, Irving Valeriano, et al. Learning-based energy consumption prediction. *Procedia Computer Science*, 203:272–279, 2022.

[36] Karan Aggarwal, Z Chenlei, J Campbell, Abram Hindle, and Eleni Stroulia. The power of system call traces: Predicting the software energy consumption impact of changes. 2014.

[37] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, page 15–21, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 225–236, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. A comprehensive study on the energy efficiency of java's thread-safe collections. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 20–31, 2016.

[40] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. jstanley: placing a green thumb on java collections. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 856–859, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. Recommending energy-efficient java collections. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 160–170, 2019.

[42] Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Seep: exploiting symbolic execution for energy-aware programming. *SIGOPS Oper. Syst. Rev.*, 45(3):58–62, January 2012.

[43] Timo Hönig, Heiko Janker, Christopher Eibel, Oliver Mihelic, and Rüdiger Kapitza. Proactive Energy-Aware programming with PEEK. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, Broomfield, CO, October 2014. USENIX Association.

[44] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of java applications from the memory perspective. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 23, USA, 2001. USENIX Association.

[45] WALA contributors. WALA: Watson Libraries for Analysis, 2024. Retrieved December 4, 2024 from https://github.com/wala/WALA.

[46] SootUp contributors. SootUp: A Framework for Java Analysis and Transformation, 2024. Retrieved December 4, 2024 from https://soot-oss.github.io/SootUp/latest/.

[47] Spoon contributors. Spoon: Analyze and Transform Java Source Code, 2024. Retrieved December 4, 2024 from https://spoon.gforge.inria.fr/.

[48] JavaParser contributors. JavaParser: A Parser and Abstract Syntax Tree Generator for Java, 2025. Retrieved June 2, 2025 from https://github.com/javaparser/javaparser/.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[50] Miles Cranmer. Interpretable machine learning for science with pysr and symbolicregression.jl, 2023.

[51] The Computer Language Benchmarks Game. Fastest programs in java - the computer language benchmarks game, 2025. Accessed: 2025-07-01.

[52] Alcides Fonseca and Guilherme Espada. Type systems in resource-aware programming: Opportunities and challenges, 2022.

[53] Anthony Canino and Yu David Liu. Proactive and adaptive energy-aware programming with mixed typechecking. *SIGPLAN Not.*, 52(6):217–232, June 2017.

[54] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.

[55] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery.