UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Título do Trabalho

Nome Completo do Aluno

Mestrado em [Designação do Mestrado]
Designação da Especialização / Perfil, se aplicável

Versão Provisória

[Dissertação orientada]/[Trabalho de Projeto orientado] por:
Profª. Doutora Nome Completo do Orientador
Prof. Doutor Nome Completo do Orientador

2024

Acknowledgments

[3]

Dedicatória.

Abstract

Energy efficiency in software development is on the rise due to the increasing demand for environmental sustainability and the need to reduce technology operating costs. Despite progress, developers often lack the tools and knowledge to effectively optimize the energy consumption of their programs. This project introduces a framework for automating the construction of energy prediction models for Java methods, standard library code and user code. The pipeline begins with the automatic generation of Java programs, each designed to exercise specific methods. The programs are executed and dynamically profiled using the PowerJoular tool for estimating energy consumption. Profiling-based features extracted on the run are exploited for training machine learning models capable of predicting energy consumption given static code attributes. The resulting models are integrated into a VSCode extension that statically estimates code snippets real-time energy consumption. The tool aims to raise awareness and support energy-aware software development practices through immediate feedback on energy consumption.

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].

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 [13].

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 programs using various collection types or developer-defined classes. These programs 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:

- **Program Generation:** A module that automatically generates a large set of Java programs 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 program 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 `methodA` calls `addAll` and `set`, 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.

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.

In this project the target language is Java, so the benchmark 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 [14, 15] 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:

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

Listing 1: Example for loop

- **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.
- **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 [16]. 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 [17, 18] 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.

Intel RAPL (Running Average Power Limit) [19] is a tool for monitoring power consumption. It utilizes Model-Specific Registers (MSRs), which are used for program execution tracing, per-

formance 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.

PowerJoular [20] 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 [21], 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 [22] 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.

2.4 Code static and dynamic analysis

Static analysis, as the name implies, analyzes the code statically, meaning it examines the code without executing it. By examining the code, static analysis tools can understand how the program will behave at runtime [23], 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, on the other hand, executes the program and observes its exact behavior without having to estimate or predict. This type of analysis leaves no doubt about memory usage, output, the path taken, how much time it took [23]. 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, dynamic analysis can be time-consuming, especially for programs that take a long time to complete.

For fast power estimation, static analysis is preferable. It analyzes code faster and is better suited for large projects with multiple dependencies, where dynamic analysis can be very difficult

to achieve due to complex setup and long execution times. Although static analysis may not be as accurate as dynamic analysis, it is still a viable solution. 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, which examines 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 the ML models, this work will utilize dynamic analysis. Dynamic analysis involves executing the code under various conditions to gather real-world energy usage data. 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 language tools to be reused [24] 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 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 [25].

With LSP, language-specific behavior can be integrated in multiple editors without reimple-

mentation 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 foundation 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 front end sometimes must 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 [26]: 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 2 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 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:

- **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.

For regression models:

- **R^2 (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 (Random Forest, Gradient Boosting), Neural Networks, Gaussian Processes, Support Vector Machines (SVM) and 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, such as energy consumption. 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 itera-

tively, 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 [27]. 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 [28], 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 [29]. 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. [30] 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.

Estrada et al. [31] 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. [32] 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 [33]. 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 [34] 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. [35], 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. [36] 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 3 collections, (Lists, Sets and Maps), they only account for three sizes of the collections (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.

In this study, Oliveira et al. [37], proposed a tool (CT+) that is capable of performing static analysis of the code and recommending changes that reduce energy consumption. It improved from previous works by taking into account more collections implementations, more operations, thread safety and support for mobile applications.

3.3 Dynamic Monitoring and Profiling Tools

In addition, SEEP [38] 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 [39], 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, 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 [40], 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 [41] 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 [20] 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 provides power estimation to platform-specific tools, such as the Intel API, the Linux RAPL interface, or a custom regression model. 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. That's how JoularJx is able to provide detailed insight into the energy consumption of Java applications.

3.4 Limitations of Existing Approaches

The reviewed solutions showed significant 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 is to create a user-friendly tool that can quickly estimate energy consumption through static analysis, make use of previously trained models, and provide developers with recommendations. The energy consumption will then be displayed to the developer, making him more aware of the program's energy usage and helping him make better decisions.

Chapter 4

Approach and Implementation

Tudo novo

As described in the previous sections, the aim of this work is to make developers aware of the energy consumption of their programs.

In order to achieve this goal, a framework was built with the objective of simplifying and automating the process of program 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, all implemented in Java and structured as Maven projects, 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 output a simple and practical tool, they 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. It is important to note that the tool serves 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 provide this insight to developers it is necessary to build a tool that can provide all of that. The tool needs to be practical, which means that integrating it in an IDE is recommended. With this the developer only needs to download an extension for an IDE and will access to the insights provided by the tool.

The tool will be a VSCode extension built using the LSP as explained in 2.5. While VSCode may not be the most commonly used IDE for Java projects compared to Eclipse or IntelliJ IDEA, it provides a much simpler and more accessible environment for developing and testing extensions. This will make it accessible to most developers wanting feedback on the energy consumption. To make it fast, it will use static analysis to parse the code into an AST, from there it is capable of analyzing the code and using an inference function it will output the estimated cost.

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 [42] 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 tool 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 [43], SootUp [44], Spoon [45] and JavaParser [46]. 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 Spoon more suitable for generating code that conforms to Java's syntactic and semantic rules, especially in complex transformation scenarios.

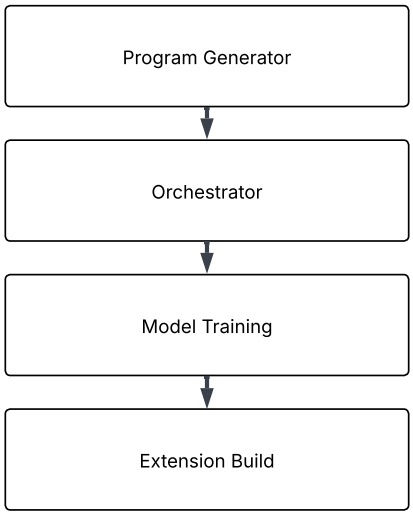

Figure 4.1: Main modules

In order to build the extension, it was necessary to build a system architecture capable of providing as final output the functioning tool. The architecture follows different stages, that were deeply analyzed before moving to the next one. The architecture is divided in four main modules, Program Generator, Orchestrator, Parser, and Tool, as illustrated in Figure 4.1.

To perform this experiment two computers were used:

The more powerful system, System 1, as described in Table 4.1, was used for program generation and data collection, as it was configured to work with SLURM (Simple Linux Utility for

Table 4.1: System Hardware Specifications

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

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 time.

4.1 Stage 1: Program Generator

In order to be able to predict energy using ML models it is of course needed a lot of data, so the models can train, and the results can be analyzed. To obtain a considerable amount of data, a program generator was built, as illustrated in Figure 4.2.

The program generator works alongside with Java Spoon to make it the more general as possible allowing custom programs to be mass generated. The generator works for custom, developer-created, Java classes, for most collections interface (Lists, Sets, Maps) implementations, for utility classes like Math, Base64, Duration, and it can be called to analyze all the public methods of the class or just specific ones.

4.1.1 Template Creation

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

The program generator starts by reading a custom Java file (or just access the Lists, Sets, Maps classes), and using Spoon it finds every public method available in the provided class. If it is needed to analyze a specific method, the generator will search in the class for every method with

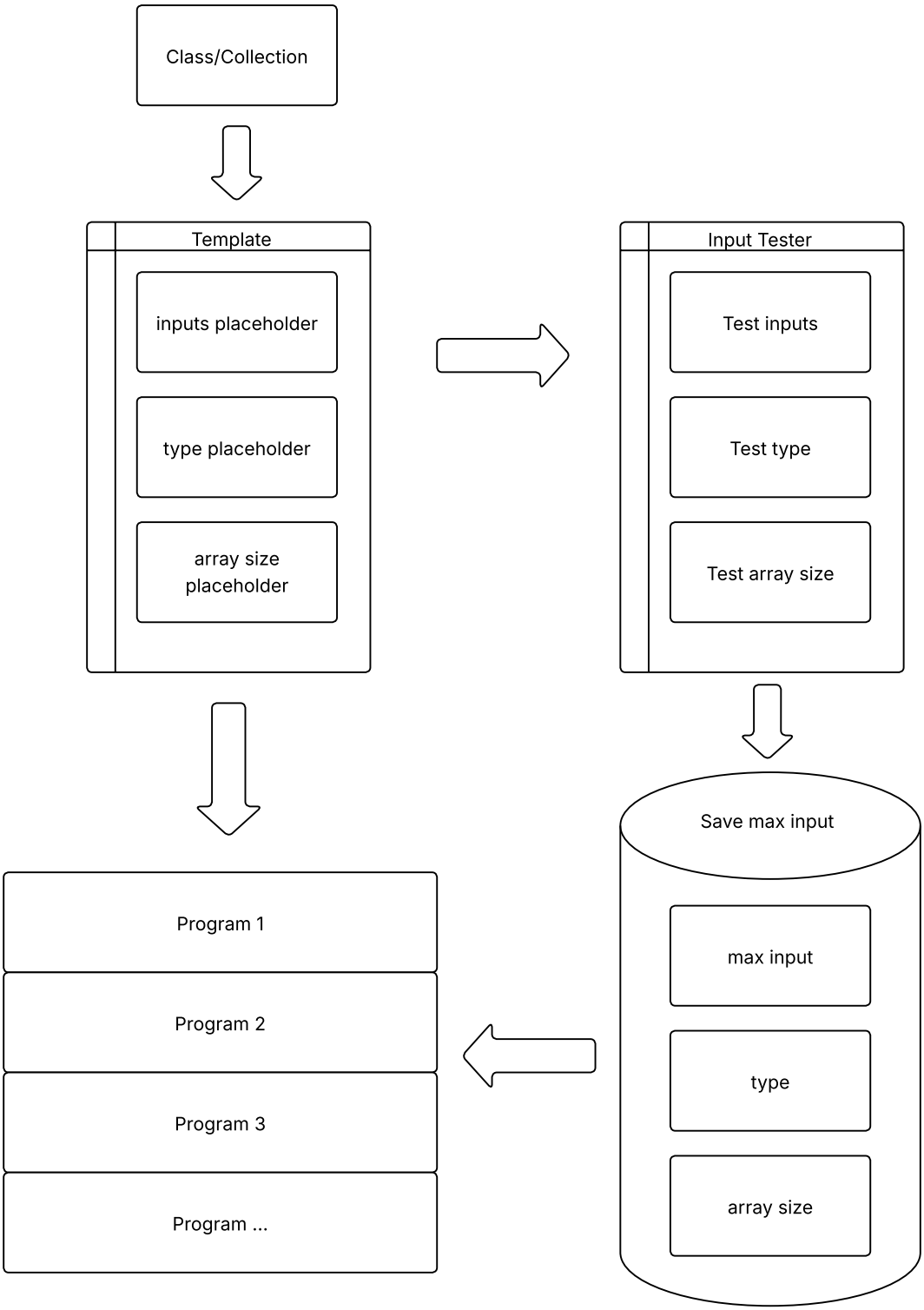

Figure 4.2: Program Generator

the name requested. Then it has access to all the public methods in the class and what parameters they receive. Since it has access to the whole custom 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.

Template important features:

- 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. (This array will be explained in 4.2).

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

```
1  static class BenchmarkArgs {
2          public ArrayList<changetypehere> var0;
3
4          public changetypehere var1;
5
6          BenchmarkArgs() {
7              this.var0 = new ArrayList();
8              CollectionAux.insertRandomNumbers(var0,
                ↪  "ChangeValueHere1_changetypehere", "changetypehere");
9              this.var1 = "ChangeValueHere2_changetypehere";
10         }
11     }
```

Listing 2: Example of variable placeholders creations

The template creates a class that holds the necessary variables the method needs to run, Listing 2 shows an example of the variables that need to be created to run the method `List.add(Object)`. First it creates the list with the smallest constructor possible, then if the variable is a collection (List, Set, Map) 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

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

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

Listing 3: Example of variable placeholders replaced

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.

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 program generation by only having to replace values of the placeholders instead of needing to create the whole program all over again, since the programs for the same method only differ in inputs, types and array size, maintaining all the structure.

4.1.2 Input Tester

A very important aspect of the program generator is the inputs it generates. Every method works differently, receives different parameters, and even when changing the values of these parameters, the method can behave completely different. So, this generator has an intermediate step, between the creation of the template and creating multiple programs, it finds the maximum size the input parameters should receive. Knowing the maximum size the different parameters can have is very important as it needs to be big enough, so the energy profiles can be abundant, but not to big so that the programs start to get out of memory or taking too much time to complete. The input search works by using binary search. It has limits on the inputs (1-100,000) and it starts by trying to run the program with the half of the max input which is 50,000. Then it runs the program for maximum of 10 seconds, which is a threshold based on empirical experimentation, and waits for

the exit code of the program, if it is an error code it will lower the input by half again, of it is a normal finish code, it will increase the input by half. This operation is done until the maximum value for the input is found. If the method that is being tested has more than one input, the input tester, first 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. This avoids having to find multiple combinations of parameters which would increase the time complexity exponentially. Since the process of finding the max inputs is time-consuming, when the maximum value is found, the values of the input type, maximum value and order (if it was the parameter 1 or parameter 2 or parameter 3) are stored in a file, so if the actual programs later generated are not stored, using this files they can be quickly generated. It is worthy to mention that the maximum inputs found depend heavily on the machine where the program generation is taking place, as different hardware will change the maximum values allowed for the inputs.

Example: 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`
- ⋮
- 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`
- ⋮
- 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`
- ⋮
- 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 range of 1 to 100,000 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 of 1 to 100,000. 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 of 100,000 was chosen due to practical experience, as larger input values would lead to higher execution times and memory crashes. Another limitation 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 1). 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. The process also has a timeout of 10 seconds. This threshold was empirically selected to balance precision and practicality, based on our needs and available hardware. Despite these limitations, the input search plays a crucial role in ensuring that the program generator produces viable test cases. By identifying input ranges that are both valid and computationally achievable, it reduces the unusable generated programs (e.g., due to timeouts or crashes), and maximizing the number of generated programs, that can be used for energy profiling.

4.1.3 Program Generation

Finally, when the templates are created, and the maximum inputs are found the program 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 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 programs being generated with random inputs, constrained by the previously identified input bounds.

This process easily generate thousands of programs which are crucial to train machine learn models that are able to predict energy consumption. When generating programs, 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 programs are ready to be compiled and used.

4.2 Stage 2: Orchestrator

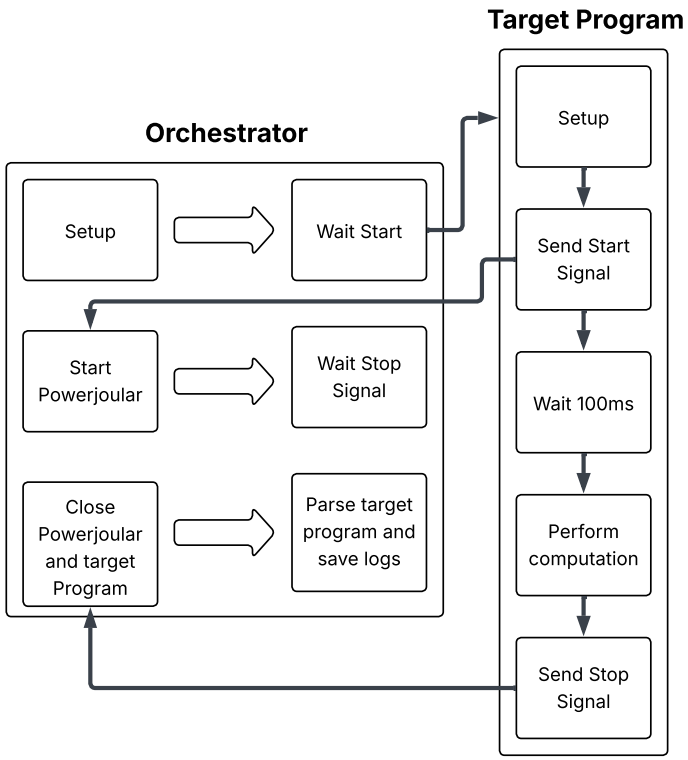

Figure 4.3: Orchestrator Workflow

Having all the programs generated, it is possible to move to the next stage. Gathering the energy profiles for each of the generated programs. This step is necessary, of course, because to give energy consumption estimates, first it is necessary to obtain energy profiles [34, 37], so the machine learning models can obtain the most accurate results possible. To make this task as easily as possible, a process was implemented, illustrated in Figure 4.3. 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. At the start some tools were tested in order to see the one that best suited the needs of the project.

Perf 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 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 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.

In our case we choose PowerJoular, it 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 used in a CSV file (easy to work with), 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 program and the measurement tool (PowerJoular) to accurately measure the energy consumption of the program 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 really difficult 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 would have too much noise to be considered. So, it surges the need to loop through the method until the tool (PowerJoular) is capable of getting its energy and then dividing the total energy by the number of times it looped. This can work, but it brings other errors that will need to be considered.

If we approach 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. But there is a problem, the methods are treated like a black box, it is not possible to know what the methods are going to do with the parameters or with its variables, for example, the method `List.size()` it receives nothing, and return the size of the list which is fine. However, this raises problems with other methods, such as `List.add(Object)`, maybe not in the first iterations. But what if the loop iterates too many times? It will make the list much bigger than the initial list, which again will cause differences in the energy measurements and later in the energy predictions. Now, suppose we have a custom 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.

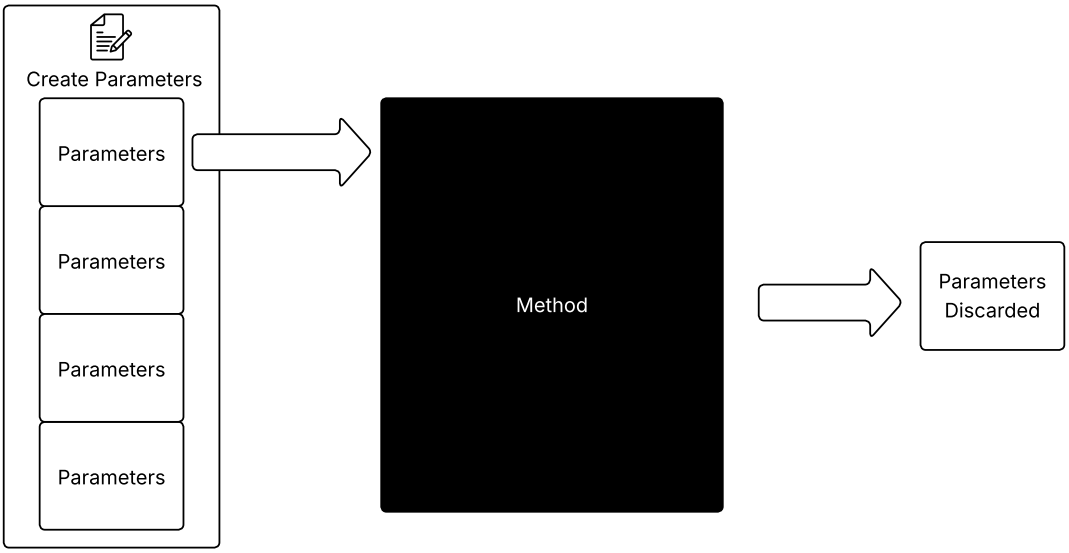

Figure 4.4: Array example

To prevent this side effect from happening the approach used was to create an array that holds the parameters, that can be seen in the Figure 4.4. Basically, the parameters are created into an array, all with the same values, but with different references. Now, if one of the elements is changed during its execution on the method, it will not affect the other's execution. Of course, since now there are multiple references of the same values, the memory usage is much higher, so that is why there is an array size, also empirically chosen, to avoid memory errors and avoid having the array so small the PowerJoular would not have time to perform the measurements. There are, for now, 3 different array sizes (75,000, 100,000, 150,000). 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. Using an array in this case is preferable to a list, as it introduces less overhead and has a smaller impact on energy measurements.

Additionally, 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.

```
1  private static int computation(BenchmarkArgs[] args, int iter) {
2          int i = 0;
3          while (!TemplatesAux.stop && i < iter) {
4                arrayList_add_java_lang_Object_(args[i].var0,
                  ↪  args[i].var1);
5                 i++;
6          }
7          return iter;
8      }
```

Listing 4: Computation method

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.

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

The workflow of this step can be described as follows:

- The orchestrator launches a command to start the target Java Program and waits a signal.
- The Java program 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 program setup. Finally, it starts PowerJoular using that PID. Then it waits for the stop signal.
- The Java program 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 program, if needed. Then it parses the target program 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 program. At the end of the process, a log file is created containing key information, including all the programs used, the PowerJoular files generated, temporary files, error logs, and features.

The features extracted from the parser can be seen in the Table 4.2

Table 4.2: Features Extracted

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

4.3 Stage 3: Model Training

Now, that all the energy profiles are collected it is possible to finally start training the models. As explained in 2.6 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.

First there is a merge of features. 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 features are stored in a new CSV and for the python program in a Data frame, to be processed.

This part was developed in python using some libraries specialized on machine learning, like scikit-learn (sklearn) [47] which contains some models and functions to evaluate the models, and PySR [48] which was already explained in 2.6 an is also already implemented.

In this phase the models tested were: Decision Tree Regressor, Random Forest, Gradient Boosting, Linear Regression, and PySR. Firstly the values of the MSE and R^2 are evaluated by the default values of sklearn, then in a second pass, GridSearch was used, which is a function of sklearn that tries to find the best parameters for a model. After that the scores and models are saved. The GridSearch does not work with PySR as they are from different libraries so, the parameters for PySR were manually set.

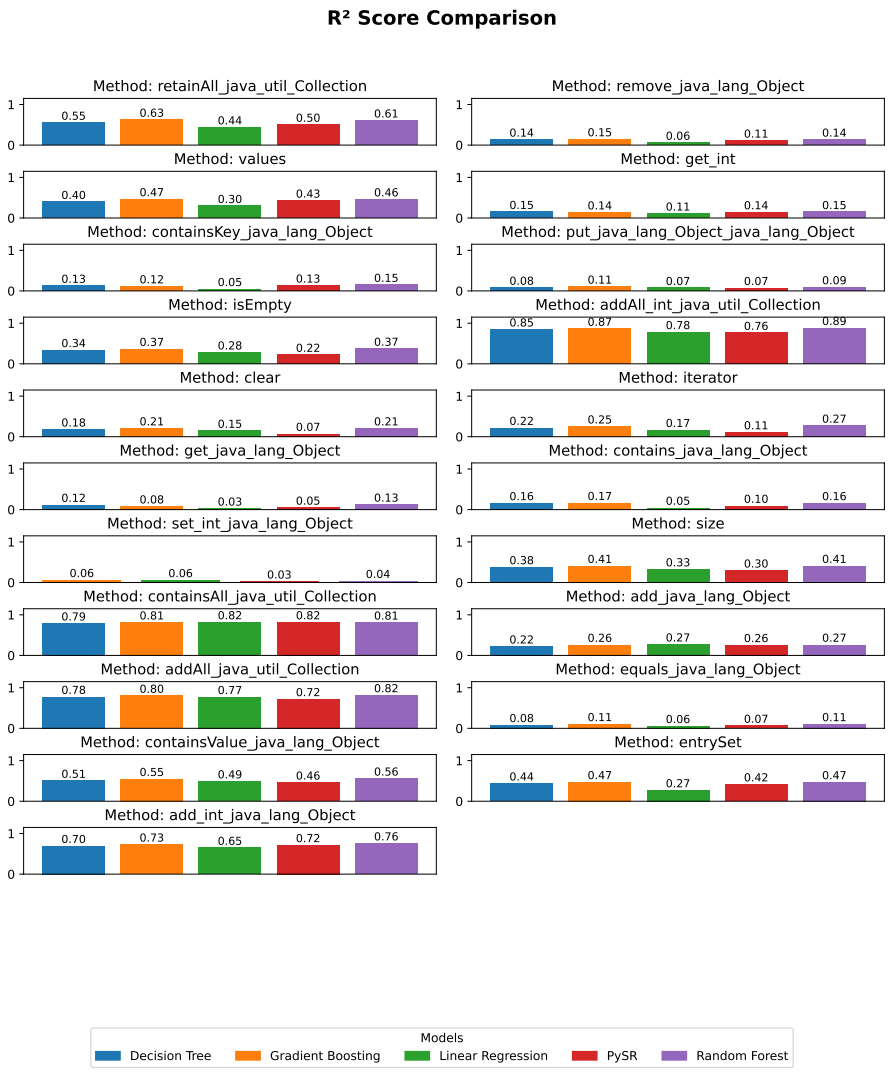

Figure 4.5: R² Comparison

Figure 4.6: MSE Comparison

The results of the R² can be seen in the figure 4.5. Ideally the best score is 1, meaning that the model can get 100% of the prediction right, however that is not possible in most cases, and this project is not an exception to the rule. It is noticeable that most values are really low (bellow 0.8), 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 R² of around .8 for most models, meaning that for those 3 methods on average the predictions will be 80% correct.

For the other methods the scores were lower. The explanation found for this result was the following:

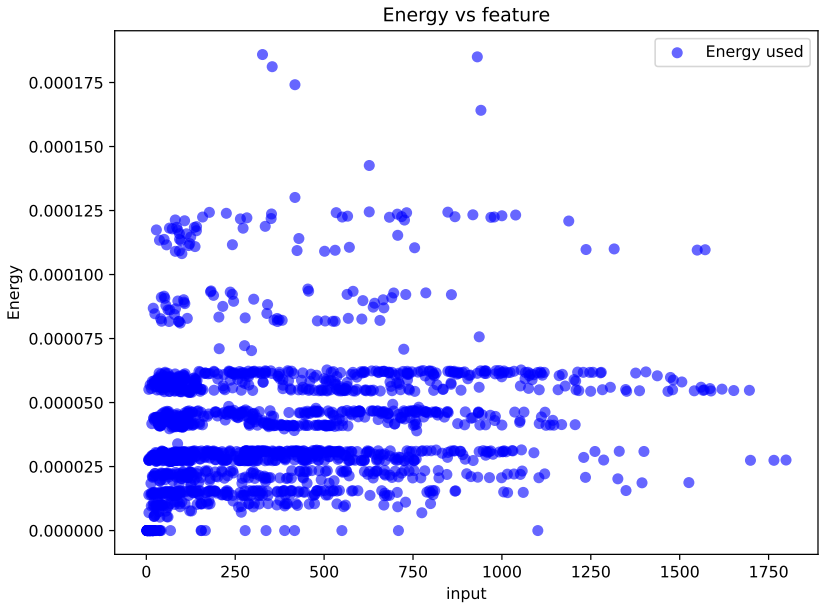

Figure 4.7: Energy for the size method with different list sizes.

The 3 methods with the most accuracy were the ones that generated bigger energy outputs, because the methods were computationally more powerful 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 had some difficulties predicting the energy. This does not mean that a lower R² will completely make the model unusable for some methods, as their MSE, that can be seen in the figure 4.6 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 R².

Another factor that may contribute to low R² 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 complete very quickly and consume very little energy, even minimal measurement noise can significantly affect the recorded values. Figure 4.7 illustrates this effect: although an increase in energy with input size is typically expected, the recorded energy values for `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.

Nonetheless, the lower R^2 scores should be addressed, and what can be done to improve the results of these models is to have the program 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.

In general, most of the models present similar scores, however the chosen one was PySR. As 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. 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.

4.4 Stage 4: Extension Build

The last stage was to build an extension for an IDE, VSCode. 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. When opening the extension side page, it contains some sliders that can change the input values, and it has the estimate button, to predict the energy. The sliders are updated when the document is saved.

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 inputs and

important variables are `list` and `i`. The extension does this to every method it finds and groups it by method. In the end it creates groups of input variables for each method, and displays it in sliders. The sliders allow the user to change the input values and when pressing the estimate button, depending on the changed values, it will change the energy estimation.

If a method calls another user-defined method that already has an assigned energy cost (but is not a trained method), 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.

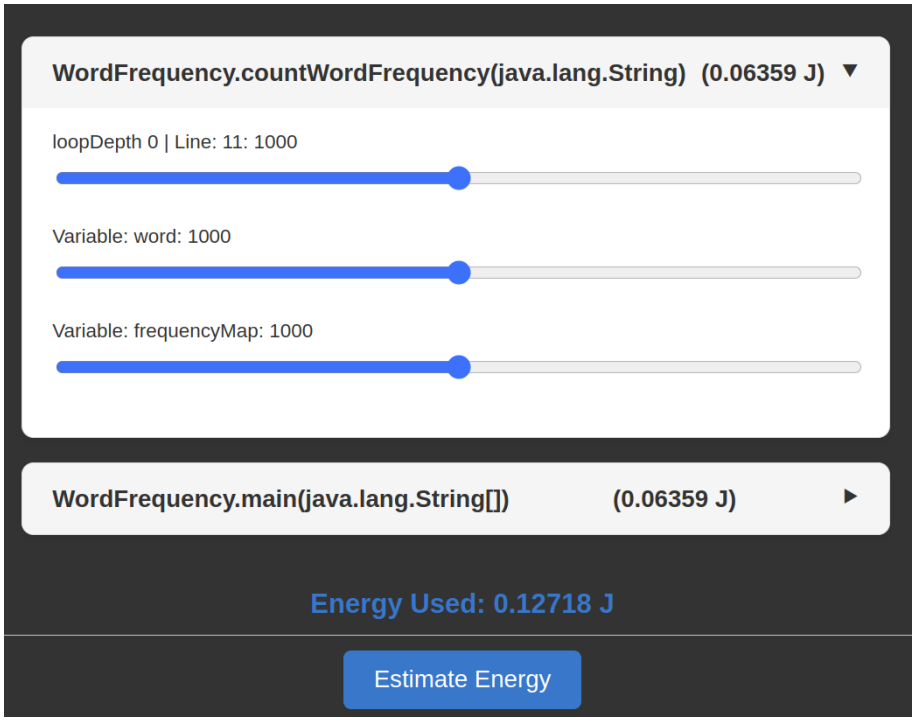

Figure 4.8: Extension example HashMap

This extension can help understand how much energy the code is using. For example, this small code snippet 5 that 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;
        frequencyMap.put(word, frequencyMap.getOrDefault(word, 0)
         ↪  + 1);
    }
    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

Using the energy prediction extension it is possible to understand how much energy this code uses, as it can be seen in the figure 4.8. In the figure 2 methods can be seen, and their energy is displayed, also one of the containers has the important variables that might affect the energy of the code. 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. The 2 variables shown in the figure4.8, show up because of the method `Map.put(Object,Object)`, which means it will have 3 inputs. The first one is the collection input, (i.e `frequencyMap`), the second one is the first Object which is the variable `word` and the last Object is not a variable, so it does not appear in the extension.

If, for instance, instead of using `HashMap`, `TreeMap` is used, the energy differences can easily be noticed 4.9. This shows a great example of how two implementations of the same collection can differ in energy. It can also be used to compare different method implementations, that achieve the same output but rely on different approaches.

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 4.10 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 2 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.

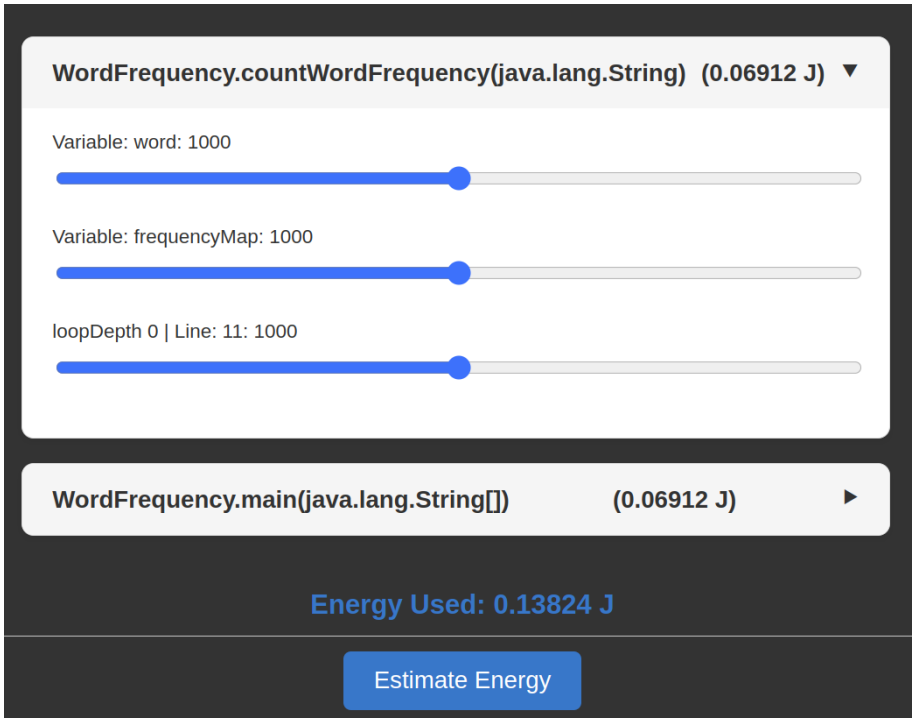

Figure 4.9: Extension example TreeMap

Figure 4.10: Expression for method Map.put(Object, Object)

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

Chapter 5

Results

Tudo novo

At the beginning of the experiment, some tools 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, a Python framework designed to facilitate experiment measurements was used [22]. 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 simpler orchestrator was developed. Although it performed the same core function as the original framework, measuring energy consumption, it was more straightforward, focusing exclusively on energy measurement rather than providing a general-purpose solution. This new orchestrator was implemented in Java.

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, two more orchestrators were implemented. A simplified custom version of the Python framework was also created. 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.

To better understand the inconsistencies, figure 5.1 details the differences. The figure contains 100 runs of the Fibonacci recursive program written in Java and order by the less energy to the highest energy. And it shows the energy reads for the four different orchestrators used. The labels contain the average energy values and its standard deviation.

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.

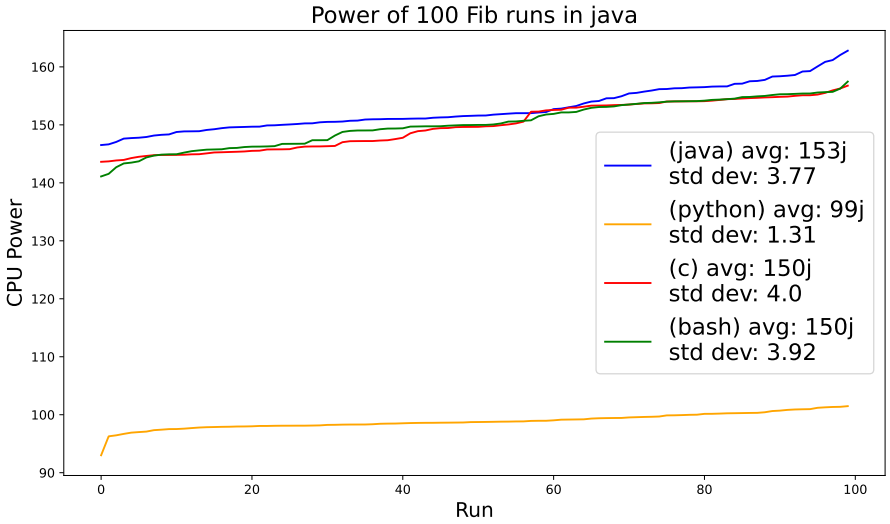

Figure 5.1: Orchestrators Comparison

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 (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.

5.1 Predicted vs. Measured Energy

Since the extension has been built, 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 4.1), 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.

```
1       ArrayList<Integer> l = new ArrayList<>();
2       ArrayList<Integer> l2 = new ArrayList<>();
3       l.addAll(l2);
```

Listing 6: Code example

To test if the extension tool is accurate, the energy measurement needs to be run in the same machine were the data was collected. For this a simple program was developed 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.

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 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, so now the values are getting off. The method `addAll(Object)` is one of the methods that has the best accuracy of around 80%, and the other two do not. This makes it so that when adding the other two methods the real energy value starts drifting away from the real one.

This can get worse when the number of methods used and variables involved start increasing, leading to greater divergence between the prediction output and actual values.

Using the example of the word counting program 5, which involves Maps and loops, the bigger gap between reality and prediction can be seen, as the Map methods predictions are not very accurate. The loop has a size of 1000 (extension default) and the Map contains 3 unique entries, since the input string includes just three distinct words.

The energy measured is in the range of 0.0027J to 0.0030J and the prediction is 0.033J. This deviates by a factor of 10 which is considerable and makes the prediction absolute values not viable. However, the relative predictions remain consistent, as input sizes increase, the predicted energy consumption also increases.

Another problem is that summing the predictions of individual methods can lead to error accumulation, which increases discrepancies between predicted and actual energy values as more methods are used, even if the individual models are accurate. To mitigate this, one potential solution is to introduce 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 while the tool provides useful estimations of energy consumption, especially in terms of relative trends between methods and input sizes, its absolute predictions can vary significantly depending on the type of operations being analyzed. Additionally, while some methods show high prediction accuracy (e.g., addAll(Object)), 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.

5.2 Limitations and Challenges

During program generation, most of the tested collections focused on List, Set, and Map. However, programs were also generated for another collection category, the Math collection. This introduced an issue. The computations in these programs were extremely fast, completing too

quickly for PowerJoular to measure any meaningful energy consumption. As a result, all energy readings for the Math collection programs were reported as 0J. 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 collections, they were too small for the Math collection. 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 program generation, large enough to ensure that each program runs for at least one second, giving PowerJoular sufficient time to measure energy consumption. However, implementing this would significantly increase the time required for program 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 programs 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.

Chapter 6

Conclusion

The techniques and tools in past and recent research have been studied, and this work improves on those different research and tools, by providing an easy-to-use tool that is also easy accessible and provides energy estimations. For now, the tool to perform the energy profiling as been selected, (PowerJoular) and a process to create the profiles has been made. This process, as explained in section **??**, allows targeting specific parts of programs, or the entire program, measure the energy consumption and extract important features. The selection of the tools had some setbacks. When testing PowerJoular it was noticed that it didn't work correctly when using python as an orchestrator, obtaining different measurements than the other orchestrators, (Java, C, Bash), as explained in section 5. Other than that, the tool worked fine, and it's capable of completing its task of energy profiling. The next step is to start the development of the energy inference function, by collecting the required data. When completed, it is expected that the tool can raise energy awareness among developers, and help facilitate the process of making energy efficient code.

Chapter 7

Future Work

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] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

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

[24] 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.

[25] 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/.

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

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

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

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

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

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

[47] 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.

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

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

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

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

[52] 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.