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 proposes an IDE extension designed to estimate the energy cost of programs and code snippets through static code analysis. The tool works by performing a static analysis of the code and feeding it to a machine learning model that can quickly estimate the energy cost. The feedback is intended to increase developers' energy awareness and encourage more energy-efficient coding practices.

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

Contents

List of Figures

List of Tables

xcolor

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.

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[**?**, **?**, **?**, **?**] 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[**?**]. This trend has become even more significant with the increased use of artificial intelligence [**?**], especially large scale models such as ChatGPT, 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 (CRAC) 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 [**?**].

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. 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[**?**]. 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[**?**].

Also, most current tools can measure the energy of programs and applications as they run (e.g., Android Studio Power Profiler [**?**]), 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[**?**].

1.2 Objectives

This thesis proposes a tool 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 an IDE extension for Vscode that integrates these functionalities, provides immediate feedback on the energy impact of Java applications, and allows developers to adjust their code to meet efficiency requirements. The tool should be easy to use, requiring minimal knowledge of energy consumption, while helping programmers understand the energy footprint of their software. This increased awareness will enable them to understand the overall impact of their coding choices on energy consumption and efficiency.

To create this tool, 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.

1.3 Contributions

This thesis presents a new tool designed to estimate the energy consumption of software applications written in Java. The tool combines static analysis techniques with machine learning to provide fast and accurate predictions, effectively capturing the non-deterministic nature of energy usage in software. It also presents a simple user interface, that allows interactions with sliders to change the predictions of the models.

1.4 Document Structure

This document is organized as follows:

- Chapter **??** – 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 **??** – 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 **??** - explains in detail the existing problem and what is the solution, and a detailed explanation on how the solution will be built.
- Chapter **??** - reports on the experiments made so far, and the obtained results.
- Chapter **??** - summarizes the work completed to date.
- Chapter **??** - outlines future research directions.

xcolor

Chapter 2

Background

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[**?**]. 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[**?**, **?**] 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.

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

PowerJoular [**?**] 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[**?**], 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[**?**] 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 **??**.

2.2 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[**?**], 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[**?**]. 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.

2.3 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[**?**]: 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 iteratively, over generations, through mutations and crossovers, that sometimes can be random, guided by a fitness function. Genetic programming is useful for tasks like symbolic regression and feature classification. However, it can be computationally expensive and can produce inconsistent results due to its uncertain nature. A good example of genetic programming in machine learning is Python Symbolic Regression (PySR). PySR builds on this technique to discover mathematical expressions that model data in a way that is both accurate and interpretable. Unlike black-box models, PySR generates human-readable formulas, making the results more interpretable and transparent. PySR works by searching the space of mathematical expressions to find those that best fit the data:

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

- Evaluates how well each expression fits the data using a loss function (like MSE).
- Expressions are then evolved over several generations using operations like mutation and crossover to produce new, often more complex formulas.

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

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

These algorithms were taken into account when developing the model that can best predict energy.

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[**?**].

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[**?**]. 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[**?**].

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 [**?**], 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[**?**]. 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. [**?**] 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. [**?**] 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. [**?**] 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[**?**]. 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 [**?**] 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.[**?**], 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. [**?**] 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. [**?**], 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 [**?**] 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[**?**], 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, **??**, 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[**?**], 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[**?**] 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[**?**] 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.

xcolor

[most]tcolorbox minted

listings, breakable

Chapter 4

Approach

As described in the previous sections, the aim of this work is to make developers aware of the energy consumption of their programs. By using a simple and practical tool, they can quickly and accurately estimate their program's 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 this 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 Language Server Protocol (LSP). 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. Although the tool is initially built for VSCode, its use of the LSP standard means it can be extended to other IDEs like Eclipse or IntelliJ IDEA. Since LSP handles the communication between the language server and the development environment, porting the tool to new IDEs would primarily involve adapting the user interface and integration specifics, rather than reworking the core logic.

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 [**?**] 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 develop-

ers 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 [**?**], SootUp [**?**], Spoon [**?**] and JavaParser [**?**]. 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.

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 4 main modules, the Program Generator, Orchestrator, Parser and Tool.

4.1 Stage 1: Program Generation

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. 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 Java classes or for some collections (Lists, Sets, Maps), and it can be called to analyze all the public methods of the class or just specific ones.

4.1.1 Template Creation

```
1  package com.generated_progs.HashMap_put_java_lang_Object_java_lang_Ob
   ↪  ject_;
2  import com.template.aux.DeepCopyUtil;
3  import com.template.aux.ArrayListAux;
4  import com.fasterxml.jackson.core.type.TypeReference;
5  import com.template.aux.TemplatesAux;
6  import java.util.HashMap;
7  public class HashMap_put_java_lang_Object_java_lang_Object_ {
8      public static void main(String[] args) throws Exception {
9          int iter = 0;
10         try {
11             HashMap<changetypehere, changetypehere> var0 = new
               ↪  HashMap();
12             ArrayListAux.insertRandomNumbers(var0,
               ↪  "ChangeValueHere1_changetypehere", "changetypehere");
```

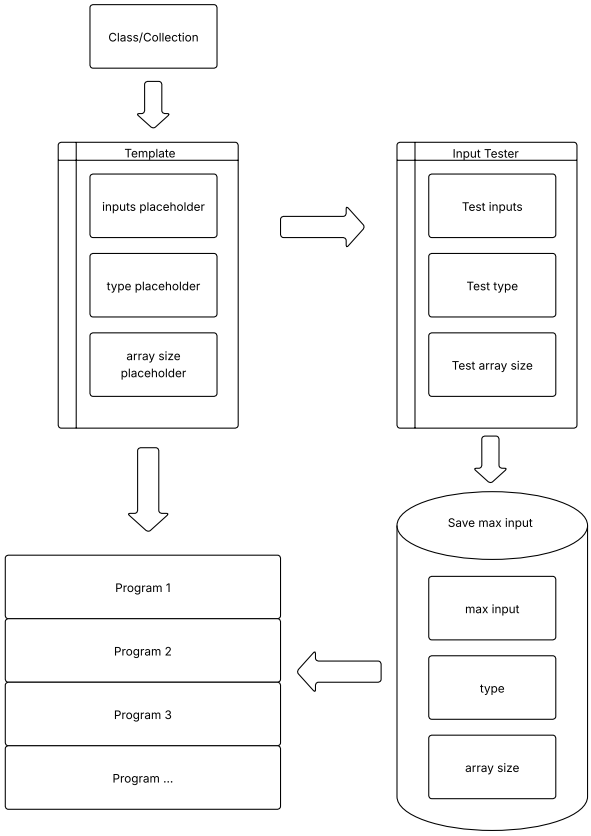

Figure 4.1: Program Generator

```
13              changetypehere var1 = "ChangeValueHere2_changetypehere";
14              changetypehere var2 = "ChangeValueHere3_changetypehere";
15              BenchmarkArgs[] arr = new
                ↪  BenchmarkArgs["numberOfFunCalls"];
16              populateArray(arr, var0, var1, var2);
17              TemplatesAux.sendStartSignalToOrchestrator(args[0]);
18              TemplatesAux.launchTimerThread(1100);
19              iter = computation(arr, arr.length);
20          } catch (OutOfMemoryError e) {
21              TemplatesAux.writeErrorInFile("HashMap_put_java_lang_Obje
                ↪  ct_java_lang_Object_", "Out of memory error caught by
                ↪  the program:\n" + e.getMessage());
22          } catch (Exception e) {
23              TemplatesAux.writeErrorInFile("HashMap_put_java_lang_Obje
                ↪  ct_java_lang_Object_", "Error caught by the
                ↪  program:\n" + e.getMessage());
24          } finally {
25              TemplatesAux.sendStopSignalToOrchestrator(args[0], iter);
26          }
27      }
28
29      static class BenchmarkArgs {
30          public HashMap<changetypehere, changetypehere> var0;
31
32          public changetypehere var1;
33
34          public changetypehere var2;
35
36          BenchmarkArgs() {
37              this.var0 = new HashMap();
38              ArrayListAux.insertRandomNumbers(var0,
                ↪  "ChangeValueHere1_changetypehere", "changetypehere");
39              this.var1 = "ChangeValueHere2_changetypehere";
40              this.var2 = "ChangeValueHere3_changetypehere";
41          }
42      }
43
44      private static void
        ↪  hashMap_put_java_lang_Object_java_lang_Object_(HashMap var,
        ↪  changetypehere arg0, changetypehere arg1) {
45          var.put(arg0, arg1);
46      }
47
48      private static int computation(BenchmarkArgs[] args, int iter) {
49          int i = 0;
50          while (!TemplatesAux.stop && i < iter) {
51                hashMap_put_java_lang_Object_java_lang_Object_(args[i].
                  ↪  var0, args[i].var1,
                  ↪  args[i].var2);
52                 i++;
53          }
54          return iter;
55      }
56
57      private static void populateArray(BenchmarkArgs[] arr,
        ↪  HashMap<changetypehere, changetypehere> var0, changetypehere
        ↪  var1, changetypehere var2) {
```

```
58            for (int i = 0;i < "numberOfFunCalls";i++) {
59              arr[i] = new BenchmarkArgs();
60            };
61        }
62
63        private String input1 = "ChangeValueHere1";
64
65        private String input2 = "ChangeValueHere2";
66
67        private String input3 = "ChangeValueHere3";
68    }
69
```

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 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 the next section).

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 (next section), so when the placeholders are replaced with actual values, the orchestrator can run the program, and communicate with them. 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.

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.

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 (timeout also defined by us) 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 param 1 or param 2 or param 3) are stored in a file, so if the actual programs later generated are not stored, using this files they can be quickly generated.

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 = 0, arg2 = 1`

Note: Since the list size (`arg0`) remains 1, the maximum valid index (`arg1`) is constrained to 0. 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.

4.2 Step 3: Implementation and testing

Once the main components of the tool are built, they need to be assembled into the extension. When using it, the developers should be able to see the total energy estimate of their code in the IDE, and it should also show the estimates for each function and its most energy consuming lines. The estimate alone may be enough to understand if the code is high or low in energy consumption, for example, if the developer has two implementations of the same function and they both give different values, it may be easy to understand which one consumes the most. However, this may not always be the case, so the tool will also provide some information to help the developer know how good or bad the energy efficiency of the code is.

Another important step is to test and ensure that the tool performs as expected on most machines, delivers the most accurate estimations possible, and undergoes a final comparison with other tools to evaluate its effectiveness.

Chapter 5

Results

In the start of the project, some tools were tested in order to see how to get the energy profiles for later use. The tools tested were PowerJoular, Powertop, Perf and JoularJx.

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 doesn't 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.

As described in the **??**, PowerJoular is the best option. As a command line program it can be easily adapted to measure any program or code snippet in most languages. So it was combined with the framework experiment-runner, that facilitates the process.

Initially, JoularJx and the experiment-runner using PowerJoular were used to explore their capabilities and familiarize with the tools. A sample Fibonacci program written in Java and C was used as a test case. However, the energy measurements provided by the two tools differed, and the experiment runner occasionally encountered errors. Later, it was determined that these problems were due to incorrect use of the framework. However, it was still decided the best approach was to make a new orchestrator similar to the Experiment-Runner, but simpler and specifically focused on measuring process energy consumption using PowerJoular. A Java-based orchestrator was initially developed because the Fibonacci implementation was also in Java. However, when tested, the energy measurement results differed significantly from those of the experiment runner, even though the main difference was the programming language (Java vs. Python).

To further analyze these inconsistencies, another orchestrator was developed in Python. This allowed for a closer examination of the differences in energy measurements and a deeper understanding of the behavior of the tools.

Using the process explained on **??** it was noticeable that the Java orchestrator was getting

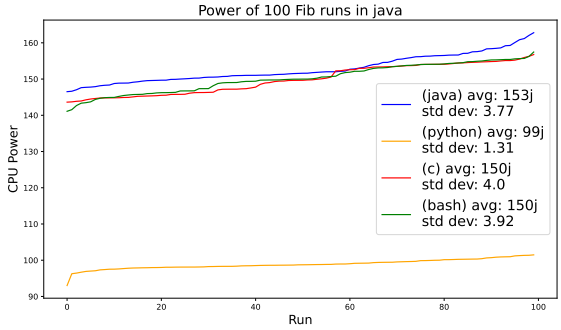

Figure 5.1: orchestrators comparison

significantly more energy consumption than the Python one, which is not very logical, since they both target the same program. So, to try and check which one was having problems, two more orchestrators were implemented, one in C and another in bash.

After running the tests again it was possible to see that the Python orchestrator was getting values way more different from the other three orchestrators as show in Figure **??**. 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.

Further analysis of the orchestrators revealed a notable difference in behavior. When the Python orchestrator was running, both the parent and child processes consumed CPU resources. In contrast, the other orchestrators (Java, C, and Bash) showed CPU usage only in the child process. This disparity may explain why PowerJoular reported lower energy consumption for the Python orchestrator. Since the CPU load was shared between the parent and child processes, PowerJoular, which measures energy only for the child process (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.

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 **??**. 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

Building on the progress made so far, additional energy profiles must be created to develop a robust energy inference function. This task involves collecting profiles from different machines and gathering enough data to effectively train the machine learning model. The static analysis tool will be implemented and used to provide important code, specifically, certain features that will feed into the model.

After that the efforts will be on developing the actual energy inference function. This task is expected to be the most time-consuming, as it is crucial to the tool functioning. It is most likely that this task overlaps the energy profiling task as more profiles might need to be collected. The task will start by implementing the ML algorithms described in **??**, test their accuracy and improving it by tuning the energy profiles for many possible cases.

When the function is completed, it will be tested against other tools to ensure its quality. The extension will then be built and tested in IDEs. While developing the function and the extension, the writing of the thesis will also be done in parallel documenting every information.

The work plan is illustrated more clearly in Figure **??**, which provides a visual representation of the described tasks. While the dates shown in the figure may not precisely align with the actual timeline, the sequence of tasks remains accurate.

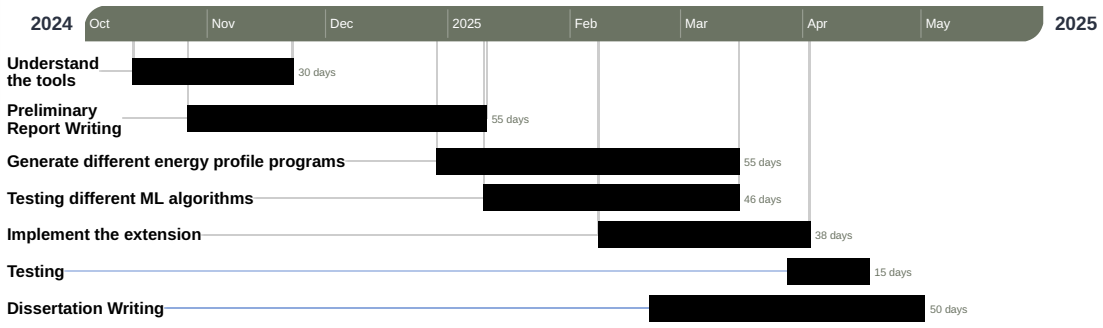

Figure 7.1: Work Plan